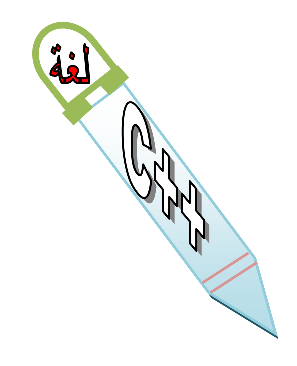


الوحدة الأولى

مقدمة إلى البرمجة بلغة ++C





بنهاية هذه الوحدة:

- ستتمكن من كتابة برامج ++C بسيطة.
- ستتمكن من استخدام عبارات الإدخال والإخراج.
- ستتعرف على أنواع البيانات الأساسية في ++C.
- ستتمكن من إستعمال العوامل الحسابية في ++C.
- ستتعرف على العوامل العلائقية في ++C.

1.1 المقدمة

تعتبر لغة ++C من أشهر اللغات التي تتمتع بطابع القوة والمرونة لإنتاج أسرع برامج وأفضلها أداءاً. وعلى الرغم من وجود العديد من لغات البرمجة الأخرى إلا أنها تفتقر شمولية لغة ++C وقوتها . فاللغة ++C تتميز بقابليتها على معالجة التطبيقات الكبيرة والمعقدة ، والقوة في صيانة البرامج المكتوبة بها مما يوفر وقتاً في تصميم البرامج وتطويرها .

تعتبر اللغة ++C امتداداً للغة C. وقد أنشأها Bjarne Stroustrup عام 1979 م، وكانت تسمى حينها C مع فئات(C with classes)، وتغير اسمها إلى ++C في العام 1983م.

تعتمد اللغة ++C أسلوب البرمجة كائنية المنحى Object Oriented Programming، والذي يعرف اختصاراً بـ (OOP)، والذي تم تطويره بسبب قيود كانت أساليب البرمجة القديمة المتمثلة في اللغات الإجرائية تفرضها على المبرمجين . ولكي نتعرف على طبيعة تلك القيود يجب أن نلقى الضوء على ما يحدث في اللغات الإجرائية.

اللغات الإجرائية:

لغات Pascal، C، Basic و Fortran وغيرها من لغات البرمجة التقليدية هي لغات إجرائية (Procedural). أي أن كل عبارة في اللغة هي عبارة عن تعليمة للحاسوب أن ينفذ شيئاً ما : أحصل على دخل أو أجمع أرقام الخ.. ... .

لذا نجد أن البرنامج المكتوب بلغة إجرائية هو عبارة عن لائحة من التعليمات. لا تبدو هنالك مشكلة مع البرامج الإجرائية الصغيرة، فالمبرمج ينشئ لائحة التعليمات ويقوم الحاسوب بتنفيذها. ولكن مع كبر حجم البرامج لا تعود لائحة من التعليمات فعالة حيث يصعب فهم برنامج يتألف من مئات من العبارات إلا إذا كانت مقسمة إلى أجزاء أصغر، لذا تم اعتماد أسلوب الدالات (Functions) والإجراءات (Procedures) كوسيلة لجعل البرامج أسهل للقراءة والفهم، حيث تمتلك كل دالة في البرنامج واجهة محددة، وتنفذ هدفاً محدداً. ولكن المشكلة ما تزال قائمة : مجموعة من التعليمات تنفذ مهاماً محددة.

و مع تزايد حجم البرامج وتعقيدها ، يظهر ضعف الأسلوب الإجرائي، حيث تصبح البرامج الضخمة معقدة إلي حد كبير. من أهم أسباب فشل اللغات الإجرائية هو الدور الذي تلعبه البيانات فيها، حيث تعطى البيانات أهمية ثانوية على الرغم من أنها هي السبب في وجود

البرامج ، ويكون التشديد على الدالات التي تعمل على هذه البيانات ، حيث يتم تعريف البيانات خارج أي دالة لكي يصبح بالإمكان الوصول إليها من كل الدالات في البرنامج، لذا غالباً ما تكون البيانات عرضة للتغيير أو التعديل الخطأ. وعلى الرغم من أن هنالك بعض اللغات كـ Pascal وC تعرف متغيرات محلية (Local)، وهي متغيرات معرفة في دالة واحدة. لكن المتغيرات المحلية غير مفيدة للبيانات المهمة التي يجب الوصول إليها من عدة دالات في البرنامج. أيضاً هناك مشكلة طريقة تخزين البيانات بسبب إمكانية عدة دالات للوصول إليها. لا يمكن تغيير ترتيب البيانات من دون تغيير كل الدالات التي تتعامل معها.

وإذا أضفنا بيانات جديدة نحتاج لتعديل كل الدالات حتى تستطيع هذه الدالات استعمال هذه البيانات الجديدة .

غالباً ما يكون تصميم البرامج الإجرائية صعباً، لأن مكوناتها الرئيسية (الدالات) عبارة عن بنية بيانات لا تقلد العالم الحقيقي جيداً .و يصعب في اللغات الإجرائية إنشاء أي نوع بيانات جديد بخلاف الأنواع المعرفة أصلاً في تلك اللغات ، لكل هذه الأسباب تم تطوير الأسلوب الكائني المنحى.

الأسلوب الكائني المنحى:-

الفكرة الأساسية وراء اللغات كائنية المنحى هي دمج البيانات والدالات التي تعمل على تلك البيانات في كينونة واحدة تسمى كائن (Object)، وعادة تزود دالات الكائن -والتي تسمى أعضاء دالية (Member functions)- الطريقة الوحيدة للوصول إلي البيانات، لذا تكون البيانات محمية من التعديلات الخطأ ويقال أن البيانات ودالاتها مغلفة (Encapsulated) في كينونة واحدة.

مميزات اللغات كائنية المنحى :

هنالك تطابق بين الكائنات في البرمجة وكائنات الحياة الفعلية، فالعديد من الكائنات الفعلية لها وضعية (خصائص يمكن أن تتغير) وقدرات (أشياء يمكن أن تقوم بها).

في ++C تسجل بيانات الكائن ووضعيته كما تتوافق أعضاءه الدالية مع قدراته، تدمج البرمجة كائنية المنحى المرادف البرمجي للوضعيات والقدرات في كينونة واحدة تسمى كائن النتيجة لذلك كينونة برمجية تتطابق بشكل جيد مع الكثير من كائنات الحياة الفعلية.

الفئات والوراثة (Inheritance):

الكائنات في OOP هي مثيلات من الفئات ، حيث يمكننا تعريف كثير من الكائنات تابعة لفئة معينة ، وتلعب دور خطة أو قالب يتم إنشاء الكائنات على أساسه ، وه ي التي تحدد

ما هي البيانات والدالات التي سيتم شملها في كائنات تلك الفئة . لذا فالفئة هي وصف لعدد من الكائنات المتشابهة. وتؤدى فكرة الفئات إلي فكرة الوراثة، حيث يمكن استعمال فئة OOP كأساس لفئة فرعية واحدة أو أكثر تسمى الفئة ا لقاعدة (Base class)، ويمكن تعريف فئات أخرى تتشارك في خصائصها مع الفئة القاعدة ولكنها تضيف خصائصها الذاتية أيضاً ، تسمى هذه الفئات المشتقة (Derived classes).

قابلية إعادة الاستعمال Reusability:

بعد كتابة الفئة يمكن توزيعها على المبرمجين لكي يستعملوها في برامجهم ، يسمى هذا الأمر قابلية إعادة الاستعمال Reusability ويزود مفهوم الوراثة ملحقاً هاماً إلي فكرة إعادة الاستعمال حيث يستطيع المبرمج أخذ فئة موجودة أصلاً ومن دون تغييرها يضيف ميزات وقدرات جديدة إليها وذلك من خلال اشتقاق فئة جديدة من الفئة القديمة.

إنشاء أنواع بيانات جديدة:-

من أهم فوائد الكائنات أنها تعطى المبرمج وسيلة لإنشاء أنواع بيانات جديدة ،كالأرقام المركبة أو الإحداثيات ثنائية الأبعاد أو التواريخ أو أي نوع من أنواع البيانات قد يحتاج المبرمج إلي استعمالها.

تعدد الأشكال والتحميل الزائد : Polymorphism and overloading

يسمى استعمال الدالات والعوامل في أساليب مختلفة وفقاً لما يتم استعمالها عليه **تعدد الأشكال**. لا تضيف اللغة *C++* إمكانية إنشاء أنواع بيانات جديدة فقط ، بل وتتيح أيضاً للمبرمج القدرة على العمل على أنواع البيانات الجديدة تلك باستعمال نفس العوامل التي تستخدمها الأنواع الأساسية كـ + أو = ويقال عندها أنه تم تحميل هذه العوامل بشكل زائد لتعمل مع الأنواع الجديدة.

1.2 كيفية كتابة برنامج بـ C++

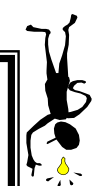
سنبدأ بكتابة برنامج يعرض نصاً على الشاشة:-

```
//Program 1-1:
//This program will display a message on the screen.
#include<iostream.h>
main ( )
{
```

```
        cout <<''welcome to C++ !\n'';
return 0;
}
```

الخرج من البرنامج:

```
welcome to C++ !
```



يقوم الحاسوب بتنفيذ البرنامج ويعود سريعاً للمحرر IDE.

من الآن فصاعداً، إذا أردت تثبت المخرجات على الشاشة عليك إضافة التالي إلى البرنامج:

```
#include <conio.h>
```

في أول البرنامج، وإضافة العبارة:

```
getch( )
```

في السطر الذي يسبق العبارة return 0.

التعليقات:Comments

```
// Program 1-1:
//This program will display a message on the screen.
```

يبدأ هذا السطر من البرنامج بالشرطة المزدوجة (//) الدالة على أن بقية السطر عبارة عن تعليق (comment)، تضاف التعليقات إلى البرامج لتساعد المبرمج أو أي شخص آخر قد يحتاج إلى قراءة البرنامج على فهم ما الذي يفعله البرنامج، لذا من المستحسن أن يبدأ كل برنامج في لغة C++ بتعليق يوضح الغرض الذي من أجله كتب البرنامج.

تستخدم الشرطة المزدوجة (//) إذا كان التعليق يمتد لسطر واحد فقط single-line comment.

هنالك نوع آخر من التعليقات يتيح لنا كتابة تعليقات تمتد إلى عدة أسطر multi-line comments ، نستطيع كتابة التعليق السابق على الصورة:

```
/* Program 1-1:
 This program will display a message on the screen
*/
```

يبدأ الرمز */ التعليق وينهيه الرمز /* . نجد أن نهاية السطر لا تعني انتهاء التعليق لذا يمكننا كتابة ما نشاء من أسطر التعليقات قبل الانتهاء بالرمز /*.

مرشدات المهيئ (Preprocessor Directive):-

```
#include<iostream.h>
```

يسمى هذا بمرشد المهيئ Preprocessor directive، وهو عبارة عن تعليمة للمصرف أن يدرج كل النص الموجود في الملف iostream.h في البرنامج، وهو ملف يجب تضمينه مع أي برنامج يحتوى على عبارات تطبع بيانات على الشاشة أو تستقبل بيانات من لوحة المفاتيح.

يسمى iostream ملف ترويسة (header file)، وهنالك الكثير من ملفات الترويسة الأخرى، فمثلاً إذا كنا نستعمل في برنامجنا دالات رياضية كـ sin( ) وcos( ) نحتاج إلى شمل ملف ترويسة يدعى math.h، وإذا كنا نتعامل مع سلاسل الأحرف سنحتاج للملف string.h. وعموماً هنالك عدد كبير من ملفات الترويسات التي يجب تضمينها على حسب طبيعة البرنامج ، تعتبر ملفات الترويسات جزء مهم من برامج لغة C++ وسنحتاج إلى شمل الملف iostream.h لتشغيل أي برنامج يقوم بعمليات إدخال وإخراج.

الدالة main :-

```
main( )
```

يبدأ تشغيل أي برنامج C++ من دالة تدعي main( )، وهي دالة مستقلة ينقل نظام التشغيل التحكم إليها. وهي جزء أساسي في برنامج C++.

الأقواس بعد main تشير إلى أن main هي عبارة عن دالة. قد يحتوى برنامج C++ على أكثر من دالة إحداهما بالضرورة هي main. يحتوى البرنامج السابق على دالة واحدة.

يبدأ تنفيذ البرنامج من الدالة main حتى لو لم تكن هي الأولى في سياق البرنامج. يتم حصر جسم الدالة main بأقواس حاصرة { } .

الخرج إلى الشاشة:-

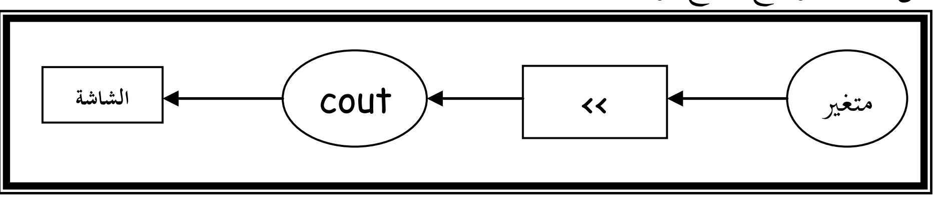
```
cout<<'' welcome to C++ !\n '';
```

هذه العبارة (statement) تجبر الحاسوب أن يظهر على الشاشة النص المحصور بين علامتي الاقتباس '' ''. ويسمى هذا النص ثابت سلسلي.

يجب أن تنتهي كل عبارة في برنامج C++ بفاصلة منقوطة ; (semi colon).

الاسم cout والذي يلفظ كـ C out يمثل كائن في C++ مقترن مع الشاشة والعامل << والذي يسمى بعامل الوضع Put to operator يجبر على إرسال الأشياء التي على يمينه إلى أي شئ يظهر على يساره.

الشكل 1-1 يوضح الخرج بواسطة **cout**.



شكل (1-1) الخرج بواسطة cout

مثال:

```
//Program 1-2: Output
#include <iostream.h>
main ( )

{
      cout  << 7 << " is an integer.\n";
      cout << 'a' << "is a character.\n";
}
```

الخرج من البرنامج:

```
7  is an integer.
a  is a character
```

من خرج البرنامج يتضح لنا الآتي:

1- يتم حصر النص المطلوب ظهوره على الشاشة بين علامتي اقتباس"**is an integer** ".

2- تتم كتابة الثوابت الرقمية بدون علامتي اقتباس 7 <<.

3- يتم حصر حرف واحد مطلوب ظهوره على الشاشة بعلامة اقتباس فردية 'a'<<.

تقوم بعض اللغات كـ Basic مثلاً بالانتقال إلى سطر جديد تلقائياً في نهاية كل عبارة خرج ، لكن **C++** لا تفعل ذلك كما أن العبارات المختلفة والموضوعة في أسطر مختلفة لا تؤدي إلى ذلك .

لا ينشئ الكائن **cout** أسطراً جديدة تلقائياً، والمخرجات في البرنامج التالي توضح ذلك:-

```
//Program 1-3:This program displays output on the screen
#include<iostream.h>
main ( )
{
 cout<<10;
 cout<<20<<30;
 return 0;
}
```

تظهر الخرج:-

```
102030
```

حيث يلتصق كل الخرج ببعضه البعض ، لذا من الجيد أن يكون لدينا طرق في ++C للتحكم بطريقة تنسيق الخرج والتي منها تتابعات الهروب(Escape Sequences).

تتابعات الهروب (Escape Sequences):

نلاحظ أنه لم تتم طباعة **n** على الشاشة ، \ تسمى الشرطة الخلفية (Back slash) أو حرف هروب (Escape character) وتسمى هي والحرف الذي يليها تتابع هروب. تتابع الهروب **n** يعني الانتقال إلى سطر جديد حيث يجبر المؤشر على الانتقال إلى بداية السطر التالي ، الآن إليك بعض تتابعات الهروب الشائعة:-

تتابع الهروب	الوصف
\n	**سطر جديد.**
\t	**مسافة أفقية.**
\b	**حرف التراجع back space.**
****	**لطباعة شرطة خلفية .**
\r	**حرف الإرجاع، يجبر المؤشر على الانتقال إلى بداية هذا السطر.**
\''	**لطباعة علامة اقتباس**

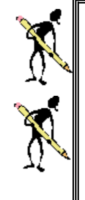
العبارة return 0 :-

تكتب العبارة return 0; في نهاية الدالة main( ). القيمة 0 تشير إلى أن البرنامج انتهى نهاية صحيحة وسيبدو لنا سبب تضمين هذه العبارة واضحا عندما نتعرف على الدوال في **C++** بالتفصيل.

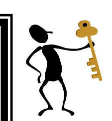
مثال آخر لبرنامج C++ :-

إليك الآن مثالاً لبرنامج يستقبل رقمين من المستخدم ويجمعهما ويعرض ناتج الجمع:-

```
// Program 1-4: Addition program
#include<iostream.h>
#include<conio.h>
main ( )   {
        int integer1, integer2, sum;
        cout <<"Enter first integer\n";
        cin >> integer1;
        cout <<"Enter second integer\n";
        cin >> integer2;
        sum= integer1+integer2;
        cout <<"sum="<<sum<<endl;
        getch();
return 0;
   }
```



```
Enter first integer
7
Enter second integer
3
sum= 10
```



١. **حدد ما إذا كانت العبارات الآتية صحيحة أم خطأ:**

- التعليقات تجبر الحاسوب على طباعة النص الذي يلي // على الشاشة عند تنفيذ البرنامج.
- تتابع الحروب **\n** يجبر المؤشر على الانتقال إلى سطر جديد.
- برنامج ***C++*** والذي يقوم بطباعة ثلاث أسطر على الشاشة يجب أن يحتوى على ثلاث عبارات تستعمل **cout**.

٢. **ما هو الخرج من العبارة الآتية:**

```
cout << "\n **\n ***\n ****\n";
```

1.3 أنواع البيانات الأساسية في لغة ++ C

هنالك سبعة أنواع بيانات أساسية في ++C ، واحد منها يمثل الأحرف وثلاثة تمثل أرقاماً كاملة (أعداد صحيحة) وثلاثة تمثل أرقاماً حقيقية. الجدول الآتي يلخص هذه الأنواع.

اسم النوع	يستعمل لتخزين	أمثلة عن القيم المخزنة
char	أحرف	''a''
short	أرقام صحيحة قصيرة	222
int	أرقام صحيحة عادية الحجم	153,406
long	أرقام صحيحة طويلة	123,456,789
float	أرقام حقيقية قصيرة	3,7
double	أرقام حقيقية مزدوجة	7,533,039,395
long double	أرقام حقيقية ضخمة	9,176,321,236,01202,6

1/ الأحرف char :-

يتم تخزين الأحرف في متغيرات من النوع char العبارة:-

```
char ch;
```

تنشئ مساحة من الذاكرة لحرف وتسميه ch. لتخزين حرف ما في هذا المتغير نكتب

```
ch='z'
```

ودائماً تكون الأحرف الثابتة كـ 'a' و'b' محصورة بعلامة اقتباس فردية.

يمكن استعمال المتغيرات من النوع char لتخزين أرقام كاملة بدلاً من أحرف ، فمثلاً يمكننا كتابة:-

```
ch=2;
```

لكن نطاق القيم الرقمية التي يمكن تخزينها في النوع char يتراوح بين -128 إلى 127 لذا فإن هذه الطريقة تعمل مع الأرقام الصغيرة فقط.

2/ الأعداد الصحيحة:

تمثل الأعداد الصحيحة أرقاماً كاملة أي قيم يمكن تعدادها ، كعدد أشخاص أو أيام أو عدد صفحات مثلاً ، ولا يمكن أن تكون الأعداد الصحيحة أرقاماً ذات نقطة عشرية ولكنها يمكن أن تكون سالبة.

هنالك ثلاثة أنواع من الأعداد الصحيحة في C++: short عدد قصير، int عدد صحيح، long طويل وهي تحتل مساحات مختلفة في الذاكرة. الجدول التالي يبين هذه الأنواع والمساحة التي تأخذها في الذاكرة ونطاق الأرقام التي يمكن أن تأخذها:

اسم النوع	الحجم	النطاق
char	1byte	-128 إلى 127
short	2byte	-32,768 إلى 32,767
int	مثل short في أنظمة 16bit ومثل long في أنظمة 32bit	
long	4byte	-2,147,483,648 إلى 2,147,483,647

3/ الأعداد الصحيحة غير المعلمة(Unsigned):-

كل الأعداد الصحيحة لها إصدارات غير معلمة (unsigned) . لا تستطيع المتغيرات التي ليس لها علامة تخزين قيم سالبة، ونجد أن نطاق قيمها الموجبة يساوى ضعف مثيلاتها التي لها علامة، الجدول التالي يبين هذا:-

اسم النوع	الحجم	النطاق
unsigned char	1byte	0 إلى 255
unsigned short	2byte	0 إلى 65,535
unsigned int	مثل unsigned short في أنظمة 16bit ومثل unsigned long في أنظمة 32bit	
unsigned long	4byte	0 إلى 4,294.967.295

4/ الأرقام العائمة (Float):

يتم استعمال الأرقام العائمة لتمثيل قيم يمكن قياسها كالأطوال أو الأوزان. ويتم تمثيل الأرقام العائمة عادة برقم كامل على اليسار مع نقطة عشرية وكسر على اليمين.

هنالك ثلاثة أنواع من الأرقام العائمة في أنظمة التشغيل الشائعة الاستعمال. وأشهر نوع أرقام عائمة هو النوع **double** والذي يتم استعماله لمعظم دالات **C++** الرياضية. يتطلب النوع **float** ذاكرة أقل من النوع **double** . الجدول التالي يوضح هذه الأنواع والحجم الذي تأخذه في الذاكرة.

اسم النوع	الحجم
float	4byte
double	8byte
long double	10byte

1.4 تعريف المتغيرات

عند كتابة أي برنامج بلغة **C++**، نحتاج لتخزين المعلومات الواردة للبرنامج في ذاكرة الحاسوب تحت عناوين يطلق عليها أسماء المتغيرات، وبما أن أنواع المعلومات المراد تخزينها تكون عادة مختلفة مثل القيم الحقيقية أو الصحيحة أو الرمزية فإننا نحتاج أن نعلم المترجم في بداية البرنامج عن أنواع المتغيرات التي نريد استخدامها فمثلاً :-

الكلمات **integer1 , integer2, sum** هي أسماء لمتغيرات عبارة عن أعداد صحيحة(النوع **int**) وهو أحد أنواع البيانات المتوفرة في **C++** .

يمكن تعريف المتغيرات في أي مكان في البرنامج لكن يجب تعريفها قبل استعمالها، يمكن تعريف المتغيرات التي تنتمي إلى نفس النوع في سطر واحد.

تسمية المتغير:

يتم تعريف المتغير بذكر الاسم ونوع البيانات التي يمكن أن يحملها هذا المتغير من أي سلسلة تحتوى على أحرف **Letters** أو أرقام **Digits** أو خطاً تحتياً **Under score(_)**، على أن لا يبدأ اسم المتغير برقم. ومن الجدير بالذكر أن لغة **C++** تفرق بين الحروف الأبجدية الصغيرة والكبيرة ، فمثلاً الأسماء **integer1 , Integer1** تعامل كمتغيرات مختلفة.

الدخل من لوحة المفاتيح:-

```
cin>>integer1
```

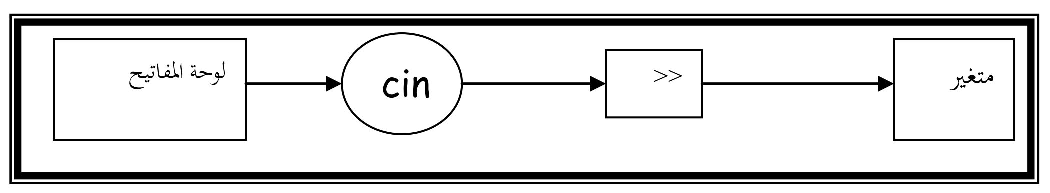
هذه العبارة تخزن الرقم الذي يكتبه المستخدم من لوحة المفاتيح في متغير يدعي integer1. يمثل الكائن cin –والذي يلفظ كـ C in– لوحة المفاتيح ، ويأخذ عامل الحصول get from (>>) الأشياء الموضوعة على يساره ويضعها في المتغير الموجود على يمينه، عند تنفيذ هذه العبارة ينتظر البرنامج أن يكتب المستخدم رقماً من النوع integer ويضغط على مفتاح Enter ، يتم تعيين القيمة التي أدخلها المستخدم إلى المتغير integer1 .

يمكن استعمال عامل الحصول عدة مرات في نفس العبارة:

```
cin >> integer1>>integer2
```

يضغط المستخدم هنا Enter، أو مفتاح المسافة Space، أو مفتاح Tab بعد كل قيمة، قبل أن يكتب القيمة التالية، ولكنه من الأفضل عادة إدخال قيمة واحدة في كل مرة لتجنب الخطأ.

الشكل(2-1) يوضح الدخل بواسطة cin.



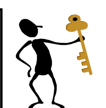
شكل (2-1) يوضح الدخل بواسطة C++

المناور endl:-

العبارة:

```
cout<<''sum= ''<<sum<<endl
```

تطبع النص =sum متبوعاً بقيمة sum ، نلاحظ أننا استخدمنا endl وهو وسيلة أخرى في C++ للانتقال إلى سطر جديد ، ويسمى مناور manipulator وendl اختصاراً لـ end line، وهو يعمل تماماً كما يعمل تتابع الهروب \n .



١. **أكتب عبارة C++ صحيحة تقوم بالآتي:**
- تعريف المتغيرات x، y ، z و result لتكون من النوع int.
- الطلب من المستخدم إدخال ثلاثة أرقام صحيحة.

٢. **حدد ما إذا كانت العبارات الآتية صحيحة أم خطأ:**
- يجب الإعلان عن المتغيرات قبل استعمالها في البرنامج.
- يجب تحديد نوع المتغيرات عند الإعلان عنها.
- لا تفرق C++ بين المتغيرات Number و number .

1.5 العوامل الحسابية (Math Operators)

لقد استعملنا عامل الجمع (+) لجمع integer1 إلي integer2، تتضمن C++ العوامل الحسابية الأربعة الاعتيادية بالإضافة إلى عامل خامس كما مبين في الجدول التالي:

العامل	الوظيفة	التعبير الجبري	التعبير في C++
+	جمع	B+h	B+h
-	طرح	B-h	B-h
*	ضرب	Bh	B*h
/	قسمة	B/h,	B/h
%	الباقي	B mod h	B%h

العوامل الأربعة الأولى تنجز أعمالاً مألوفة لدينا، أما عامل الباقي % المسمى أيضاً المعامل modulus، يتم استعماله لحساب باقي القسمة لعدد صحيح على عدد آخر، لذلك فالتعبير 20%3 يساوى 2 . تسمى هذه العوامل الحسابية بالعوامل الثنائية لأنها تعمل على قيمتين.

يمكن استعمال أكثر من عامل في تعبير رياضي واحد، فمثلاً التعبير:

```
C=(f-32)*5/9;
```

يحول درجة الحرارة من مئوية إلى فهرنهايت. (استعملت الأقواس لكي يتم تنفيذ الطرح أولاً بالرغم من أولويته المتدنية، يشير المصطلح أولوية Precedence إلى ترتيب تنفيذ العوامل، العاملان * و / لهما أولوية أعلى من +و-).وهذا ما سنراه لاحقاً بعد أن نتعرف على بقية عوامل C++ .

1.6 العوامل العلائقية (Relational Operators)

تقارن العوامل العلائقية قيمتين، وتؤدى إلى نتيجة صحيح/خطأ وفقاً لما إذا كانت المقارنة صحيح/خطأ. هنالك ستة عوامل علائقية مبينة في الجدول أدناه:

الرمز	**المعنى**	**مثال**
==	يساوى	a==b
!=	لا يساوى	a!=b
>	أكبر من	a>b
<	أصغر من	a<b
>=	أكبر من أو يساوى	a>=b
<=	أصغر من أو يساوى	a<=b

تكون التعابير المبينة في عمود المثال صحيحة أو خطأ وفقا لقيم المتغيرين a و b.

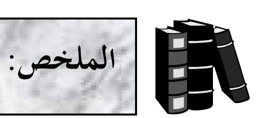
فلنفرض مثلاً أن:

- a يساوى 9
- وb يساوى 10.

التعبير a==b خطأ.

التعبير a!=b صحيح وكذلك التعبيرين a<b و a<=b ،

والتعبيرين a>b و a>=b خطأ..



الملخص:

- ♦ تبدأ التعليقات في **C++** والتي تتكون من سطر واحد بشرطة مزدوجة (//).
- ♦ تبدأ التعليقات في **C++** والتي تمتد لعدة أسطر بالرمز */ وتنتهي بالرمز /*.
- ♦ السطر #include<iostream.h> يسمى "مرشد المهيئ" وهو عبارة عن تعليمة للمصرف أن يضمن الملف iostream.h في البرنامج والذي يجب تضمينه في أي برنامج يقوم بعمليات إدخال وإخراج.
- ♦ يبدأ تنفيذ برنامج **C++** من الدالة ()main.
- ♦ المتغيرات في **C++** يجب الإعلان عنها قبل استعمالها.
- ♦ يتم تعريف المتغيرات في **C++** بذكر اسمها ونوع بياناتها وتكون الاسم من أي سلسلة تحتوى على أحرف أو أرقام أو خطاً تحتياً (_) على أن لا يبدأ اسم المتغير برقم.
- ♦ **C++** حساسة تجاه الأحرف ونعنى بذلك أنها تفرق بين الحروف الأبجدية الصغيرة (small) والكبيرة (capital).
- ♦ يرتبط كائن الخرج cout مع الشاشة وهو يستخدم في إخراج البيانات.

الأسئلة

1-أكتب عبارة *C++* صحيحة تقوم بالآتي:

- توضيح أن برنامجاً ما سيقوم بحساب حاصل ضرب ثلاثة أرقام صحيحة.
- الطلب من المستخدم إدخال ثلاثة أرقام صحيحة.
- إدخال ثلاثة أرقام عن طريق لوحة المفاتيح وتخزين قيمها في المتغيرات x، y و z.
- حساب حاصل ضرب الأرقام المخزنة في المتغيرات x، y و z وتعيين النتيجة للمتغير result.
- طباعة العبارة " :The product is " متبوعة بقيمة المتغير result.
- إرجاع قيمة من الدالة main لتوضيح أن البرنامج انتهى بنجاح.

2- إستعمل العبارات في السؤال السابق لكتابة برنامج بلغة *C++* كامل يقوم بحساب حاصل ضرب ثلاثة أرقام صحيحة.

3- حدد ما إذا كانت العبارات الآتية صحيحة أم خطأ:

أ. تمتلك العوامل الحسابية + ، - و % نفس درجة الأولوية.

ب. برنامج *C++* والذي يقوم بطباعة ثلاث أسطر على الشاشة يجب أن يحتوى على ثلاث عبارات تستعمل cout.

4-أكتب برنامجاً يستقبل من المستخدم عدداً مكوناً من خمسة أرقام ثم يقوم بطباعة الأرقام المكونة للعدد تفصلها مسافة فمثلاً إذا أدخل المستخدم العدد 13456 يكون الخرج من البرنامج

1 3 4 5 6

5- ما هو نوع البيانات الذي ستستعمله على الأرجح لتمثيل رقم موظف تسلسلي من 4 أعداد.

6- أي من العبارات الآتية تعطي المخرجات التالية:

```
1	2
2	4
```

1- cout << " 1\t2\t\n3\t4\n";

2- cout <<'1' << '\t' << '2' << '\n' <<'3' <<'\t' <<'4' <<'\n';

3- cout << "1 \n 2\t 3\n 4\t";

4- cout <<1 << '\t' << 2 << '\n' <<3 <<'\t' <<4 <<'\n';

7- أكتب جزء من برنامج يقوم بما يلي:

- ينشئ متغيرين num و denom يمثلان البسط والمقام في كسر.
- يطلب من المستخدم تزويد قيم البسط والمقام.
- يضع هذه القيم في متغيرات.
- تعرض الكسر مع شرطة (/) بين الرقمين.

قد يبدو الخرج كالآتي:

```
Enter the numerator: 2
Enter the denominator: 3
Fraction = 2/3
```

2.0

الوحدة الثانية

بنيات التحكم (I) - Control Structures(I)



بنهاية هذه الوحدة :

- ستتمكن من استعمال تعبير الإختبار **if**.
- ستتمكن من استعمال تعبير الإختبار **if… else**.
- ستتمكن من استعمال تعبير الاختبار **switch**.

2.1 مقدمة

عادة يتم تنفيذ العبارات حسب تسلسل ورودها في البرنامج ويسمى هذا بالتنفيذ التتابع ي (Sequential Execution). لكننا سنتعرض لبعض عبارات C++ والتي تجعل التنفيذ ينتقل لعبارة أخرى قد لا تكون التالية في تسلسل البرنامج، ويسمى هذا بنقل التحكم Transfer of control.

تنقسم بنيات التحكم في C++ إلى قسمين: بنيات التحكم الشرطية وسنفرد هذه الوحدة لتوضيحها. والنوع الثاني وهو بنيات التحكم التكرارية والتي سنفرد الوحدة التالية للحديث عنها.

2.2 بنيات التحكم الشرطية

2.2.1 العبارة if

أسهل طريقة لاتخاذ قرار في C++ هي بواسطة العبارة if .

مثال:-

```
//Program 2-1:
#include <iostream.h>
main ( )
{
int  num1 , num2;
cout  << " Enter two integers, and I will tell you\n"
        <<" the relation ships they satisfy:  ";
cin  >> num1>> num2;
if  (num1== num2)
        cout << num1 << " is equal to " << num2 << endl;
if (num1!= num2)
        cout << num1 << " is not equal to " << num2 << endl;
if (num1< num2)
        cout << num1 << " is less than " << num2 << endl;
if (num1> num2)
        cout << num1 << " is greater than " << num2 << endl;
```

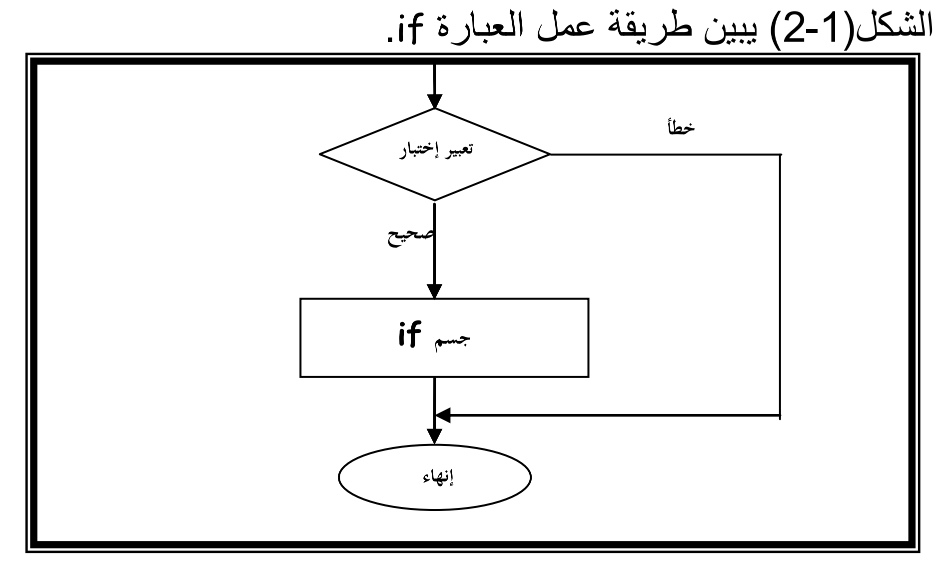
```
if (num1<= num2)
       cout << num1 << " is less than or equal to " << num2
<< endl;
if (num1>= num2)
  cout << num1 << " is greater than or equal to " << num2
        << endl;
return 0;
}
```

الخرج من البرنامج بافتراض أن المستخدم قد أدخل الأرقام num1 =3 ، num2= 7.



```
Enter two integers , and I will tell you
                    The relation ships they satisfy: 3    7
3 is not equal to 7
3 is less than  7
3 is less than or equal to  7
```

تتألف العبارة if من الكلمة الأساسية if، يليها تعبير اختبار بين قوسين، ويتألف جسم القرار الذي يلي ذلك إما من عبارة واحدة ، أو من عدة عبارات تحيطها أقواس حاصرة { }
الشكل(2-1) يبين طريقة عمل العبارة if.



شكل (1–2) طريقة عمل العبارة if

2.2.2 العبارة if…else

في العبارة if البسيطة يحدث شئ إذا كان الشرط صحيحاً، لكن إذا لم يكن كذلك لا يحدث شئ على الإطلاق . لكن لنفترض أننا نريد حدوث شئ في الحالتين إذا كان الشرط صحيحاً وآخر إذا لم يكن كذلك، لتحقيق ذلك نستخدم العبارة if... else

مثال:-

```
//Program 2-2:
#include <iostream.h>
main ( )
{
int grade ;
cout << " Enter the grade";
cin >>grade;
if(grade>= 50)
cout<<"pass" <<endl;
else
cout <<"fail"<<endl;
return 0;
 }
```

الخرج من البرنامج بافتراض أن المستخدم قد أدخل grade = 90



```
Enter the grade  90
Pass
```

أيضاً هنا يمكن أن يتألف جسم if أو else من عدة عبارات تحيطها أقواس حاصرة. الشكل(2-2) يبين طريقة عمل العبارة if…else.

هنالك طريقة أخرى للتعبير عن المثال السابق وذلك باستخدام ما يسمى بالعامل المشروط:

```
cout<<(grade>= 50 ? ''pass'' :''fail'') << endl;
```

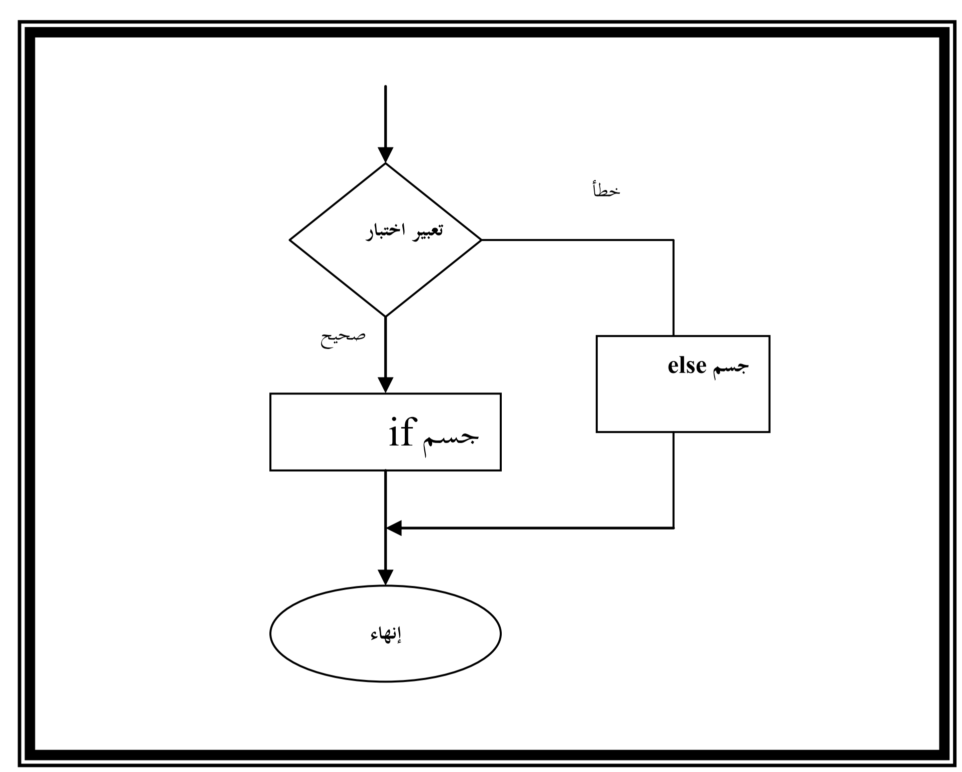
العامل المشروط هو العامل الوحيد في C++ الذي يعمل على ثلاثة قيم ويتألف من رمزين علامة استفهام ونقطتين .

أولاً يأتي تعبير الاختبار، ثم علامة الاستفهام، ثم قيمتان تفصلهما نقطتان. إذا كان تعبير الاختبار صحيحاً ينتج التعبير بأكمله القيمة الموجودة قبل النقطتين وإذا كان تعبير الاختبار خطأ ينتج التعبير بأكمله القيمة التي تلي النقطتين.

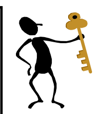
ال مثال التالي يحسب القيمة المطلقة (Absolute value) وهي تساوي سالب العدد إذا كان العدد أقل من الصفر وتساوي موجب العدد إذا كان العدد أكبر من الصفر.

```
Abs_value =(n<0) ? -n:n;
```

النتيجة هي n- إذا كان n أقل من 0 و n في الحالة الأخرى.



شكل (2-2) طريقة عمل **if...else**



ما هو الخطأ في الآتي ؟

```
if (gender==1)
 cout<<women <<endl;
else
 cout <<man<<endl;
```

العبارات if ... else المتداخلة:-

يمكن وضع العبارات ......else if ضمن بعضها البعض ، البرنامج التالي يوضح ذلك:

```
//Program 2-3:
#include <iostream.h>
main ( )
{
int grade;
cout <<"Enter the grade:" ;
cin >> grade;
if(grade>= 75)
cout<<'A'<< endl;
else
if(grade>= 65)
cout<<'B'<< endl;
else
if(grade>= 55)
cout<<'C'<< endl;
else
if(grade>= 40)
cout<<'D'<< endl;
else
cout<<"fail"<<endl;
return 0;
}
```

تنتهي العبارات المتداخلة في الجسم else وليس في الجسم if ، يمكن أن تحدث مشكلة عندما نضع العبارات ......else if ضمن بعضها البعض. فمثلاً المفروض من العبارات التالية أن تعرض الكلمة infant عندما يكون عمر الشخص أقل أو يساوى 2:-

```
if (age >2)
if (age<18)
cout <<"\n child";
else
```

```
cout <<"\n infant";
```

ولكن هنا لن يحدث ، ستظهر الكلمة infant كلما كان العمر أكبر أو يساوى 18 وذلك لأن الجزء else يتبع أقرب عبارة if إليه والتي ليس لها جزء else خاص بها. لذا إذا كنا نريد جزء else تابع لعبارة if غير موجودة قبله مباشرة علينا حصر العبارة if الموجودة بينهما بأقواس حاصرة .

```
if (age >2)
{
if (age<18)
cout <<"\n child";
} else
cout <<"\n infant";
```

2.2.3 العبارة switch

تأخذ جملة switch في C++ الشكل العام التالي:-

```
Switch (Variable name)
{
case constant1 : statement1; break;
case constant2 : statement2; break;
.
.
case constant n : statement n; break;
default : last statement;
}
```

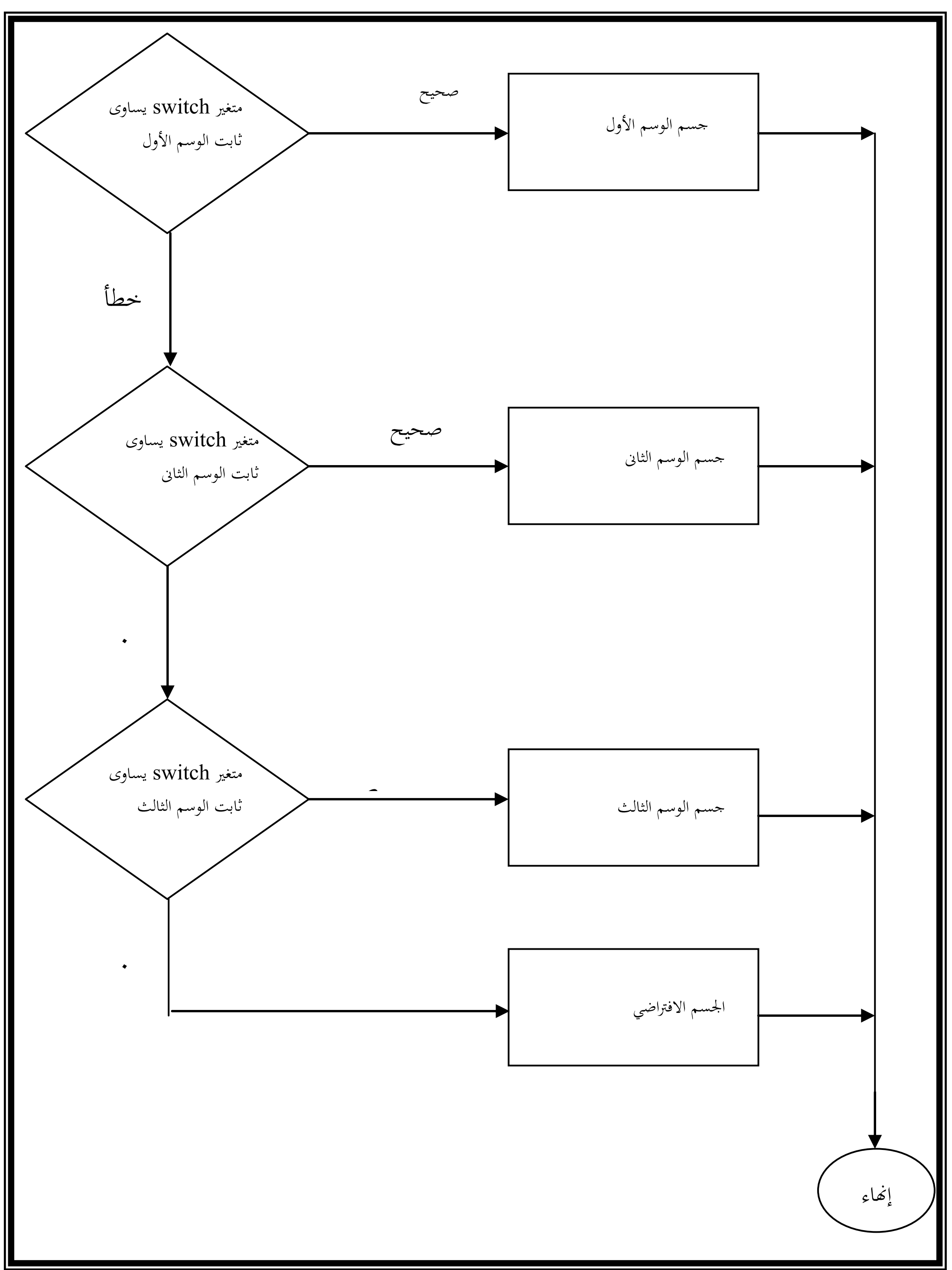
تتألف العبارة switch من الكلمة الأساسية switch يليها اسم متغير بين قوسين، ثم جسمها بين أقواس حاصرة ، تفحص العبارة switch المتغير وتوجه البرنامج نحو أقسام مختلفة وفقاً لقيم ذلك المتغير.

يتضمن جسم العبارة switch عدداً من الوسوم وه ي أسماء تليها نقطتان. تتألف هذه الوسوم من الكلمة الأساسية case ثم ثابت ثم نقطتين.

عندما تكون قيمة متغير العبارة switch مساوية للثابت المذكور في أحد وسوم case ينتقل التنفيذ إلى العبارات التي تلي ذلك الوسم وتؤدى العبارة break إلى منع تنفيذ بقية العبارة switch، وإذا لم تتطابق قيمة متغير العبارة switch مع أي وسم ينتقل التنفيذ إلى الوسم الافتراضي default .

سنقوم بكتابة برنامج لحساب عدد حروف العلة (vowels letters) وهي (a, e, i, u, o) في نص مدخل من لوحة المفاتيح . يقوم البرنامج بفحص الحرف المدخل فإذا كان الحرف a تتم إضافة 1 إلى acounter والذي تم تمهيده عند 0 . أما إذا كان الحرف المدخل e فتتم إضافة 1 إلى ecounter وهكذا بالنسبة لـ u و i و o ، إذا لم يكن الحرف المدخل حرف علة يتم تنفيذ الوسم الافتراضي والذي يقوم بإضافة 1 لـ OtherLettersCounter.

الشكل (2-3) يقوم بتوضيح طريقة عمل العبارة switch.



شكل (3-2) – طريقة عمل العبارة switch

```
//Program 2-4:
#include <iostream.h>
enum vowels{a='a',u='u',i='i',o='o',e='e'};
main( )
{
char ch ;
int acounter=0,ecounter=0,icounter=0;
int ucounter=0,ocounter=0,otherletterscounter=0;
while(cin>>ch)
switch(ch) {
case a:
++acounter;
break;
case e:
++ecounter;
break;
case i :
++icounter;
break;
case o:
++ocounter;
break;
case u:
++ucounter;
break;
default:
++ otherletterscounter;
};
cout<<endl;
cout<<endl;
cout<<endl;
cout <<"acounter: \t"<<acounter<<" \n";
cout<< "ecounter: \t"<<ecounter<<" \n";
```

```
cout<< "icounter: \t"<<icounter<<" \n";
cout<< "ocounter: \t"<<ocounter<<" \n";
cout<< "ucounter: \t"<<ucounter<<" \n";
cout<<"otherletterscounter: \t"<<otherletterscounter
     <<" \n";
return 0;
}
```

الخرج من البرنامج بافتراض أن النص المدخل
"youareverypunctional"

```
acounter:       2
ecounter:       2
icounter:       1
ocounter:       2
ucounter:       2
OtherLettersCounter:       11
```

الملخص:

♦ تأخذ العبارة if الشكل العام التالي:

```
if (Condition)
statement;
```

إذا كان جسم if يتكون من أكثر من عبارة تأخذ Statement الشكل التالي:

```
{ Statement 1;
  Statement 2;
   .
   .
 Statement n}
```

♦ تستعمل العبارة if في لغة C++ لتنفيذ عبارة أو عدة عبارات إذا كان الشرط الذي يليها صحيحاً .

♦ تأخذ العبارة if…else الشكل العام التالي:

```
if(Condition) Statement 1;
 else
Statement 2;
```

إذا كان جسم if و else يتكون من أكثر من عبارة فإننا نحيط تلك العبارات بأقواس حاصرة { }.

♦ تستعمل العبارة if… else لتنفيذ عبارة أو عدة عبارات إذا كان الشرط الذي يلي العبارة if صحيحاً ، وتنفيذ عبارة أخرى أو عدة عبارات إذا لم يكن كذلك.

♦ العامل المشروط هو وسيلة للتعبير عن العبارة if…else .

♦ العبارة switch تأخذ الشكل العام التالي:

```
switch (Variable name)
{
   case constant 1: statement 1;
   .                break;
   .
  case constant n: statement n;
                   break;

default: last statement;
}
```



الأسئلة

1/ أكتب عبارة ++C تؤدى التالي:

- إدخال قيمة متغير صحيح x باستعمال cin و >> .
- إدخال قيمة متغير صحيح y باستعمال cin و >>.
- تمهيد قيمة متغير صحيح i عند 1.
- تمهيد قيمة متغير صحيح power عند 1.
- ضرب قيمة المتغير x في المتغير power وتعيد النتيجة للمتغير power.
- زيادة قيمة المتغير y بـ 1.
- اختبار ما إذا كانت قيمة المتغير y أقل من أو تساوي x.
- طباعة قيمة المتغير power.

2/ بافتراض أن 9 = x و 11 = y ما هي مخرجات الجزء التالي من البرنامج:

```
if ( x < 10)
if ( y > 10)
      cout << "* * * * *" << endl;
      else
      cout << "# # # # #" << endl;
      cout << "$ $ $ $ $" << endl;
```

3.0 الوحدة الثالثة

بنيات التحكم(II) – Control Structures(II)



الأهداف:

بنهاية هذه الوحدة:

- ستتمكن من استعمال عوامل التزايد Increment والتناقص Decrement والعوامل المنطقية Logical operators .
- ستتمكن من استعمال حلقات التكرار **while** و **do** و **for** لتكرار تنفيذ عبارات في برنامجك.

3.1 عوامل التعيين الحسابي

باستعمال عوامل التعين الحسابي يمكن إعادة كتابة تعبير مثل:

x=x+2

على النحو

x+=2

يأخذ عامل التعين الحسابي =+ القيمة الموجودة على يمينه ويضيفها إلى المتغير الموجود على يساره. هنالك تعين حسابي لكل من العوامل الحسابية:-

a+= b	⟶	a= a+ b
a-= b	⟶	a= a- b
a*= b	⟶	a= a* b
a/= b	⟶	a= a/ b
a%= b	⟶	a= a% b

مثال:

```
//Program 3-1:
#include<iostream.h>
main ( )
{
int n;
cin >> n;
cout<< " n after adding 2 = " << a+= 2 <<endl;
cout<< " n after a subtracting 2 = " << a-= 2 <<endl;
cout<< " n after dividing by  2 = " << a/= 2 <<endl;
cout<< " n after multiplying by  2 = " << a*= 2 <<endl;
cout<< " n mod  2 = " << a %= 2 <<endl;
return 0;
}
```

الخرج من البرنامج إذا أدخلنا n =10



```
10
n after adding 2 = 12
n after a subtracting 2 = 8
n after dividing by  2 = 5
n after multiplying by  2 = 20
n mod  2 = 0
```

3.2 عوامل التزايد والتناقص

هناك دائماً حاجة في البرمجة إلى زيادة 1 أو طرح 1. هذه الحالات شائعة لدرجة أن C++ تتضمن عاملين خاصين ينفذان هذه المهمة، يقوم عامل التناقص (--) بطرح 1 من المتغير ويضيف عامل التزايد (++) 1 إليه ، المثال الآتي يبين طريقة الاستعمال:-

++a

a++

معناه إضافة 1 إلى a ، ويمكن كتابته بصورة مكافئة على النحو a=a+1

وبالطريقة نفسها يمكن إنقاص 1 من قيمة a على النحو --a أو a-- وهو يكافئ a=a-1.

ومما يجب التنبيه إليه هنا أن هنالك فرق بين ++a و a++ فعلى الرغم من كليهما يجمع 1 إلى a إلا أنه عند استعمال a++ تستخرج قيمة التعبير باستعمال قيمة a الحالية قبل زيادتها وينطبق هذا أيضاً على --a و a-- .

```
//Program 3-2:
#include<iostream.h>
main ( )
{
int c;
c = 5;
cout << c << endl;
cout << c++ <<endl;
cout << c <<endl;
c=5;
cout << c << endl << endl;
cout << ++c << endl;
```

```
cout << c << endl;
return 0;
//Continued
}
```

الخرج من البرنامج:

```
5
5
6

5
6
6
```

3.3 العوامل المنطقية

يمكن العمل على القيم صحيح/خطأ بواسطة العوامل المنطقية ، هنالك ثلاثة عوامل منطقية في ++C هي Not,Or,And كما موضح في الجدول أدناه:-

العامل المنطقي	معناه	مثال
&&	(and) (و)	x>0 &&x<10
||	(or) (أو)	x= = ||x= = 0 1
!	(not) (نفى)	!x

يكون التعبير and صحيحاً فقط إذا كان التعبيرين الموجودان على جانبي العامل && صحيحين بينما يؤدى العامل or إلى نتيجة صحيحة إذا كان أحد التعبيرين أو كليهما صحيحاً. العامل not (!) يبطل تأثير المتغير الذي يليه لذا التعبير !x صحيح إذا كان المتغير x خطأ وخطأ إذا كان x صحيحاً.

أولوية العوامل (Operator Precedence):-

يتم تنفيذ عمليات الضرب والقسمة في التعابير الرياضية قبل عمليات الجمع والطرح . في التعبير التالي مثلاً :

10*10+2*3

يتم ضرب 10*10 ثم يتم ضرب 3*2 وبعدها يتم جمع نتيجتي الضرب مما يؤدى إلى القيمة 100+6=106.

يتم تنفيذ عمليات الضرب قبل الجمع لأن العامل * له أولوية أعلى من أولوية العامل + .

نجد أن أولوية العوامل مهمة في التعابير الرياضية العادية كما أنها مهمة أيضاً عند استعمال عوامل **C++** المختلفة ، الجدول التالي يبين ترتيب أولويات العوامل في **C++** من الأعلى إلى الأدنى.

العوامل	أنواع العوامل	الأولوية
* , / , %	مضاعفة	أعلى
+ , -	جمعية	
<, > , <=, >=, ==, !=	علائقية	
&& || !	منطقية	
=	تعيين	أدنى

.43 بنيات التحكم التكرارية

3.4.1 الحلقات (LOOPS)

توفر **C++** عدداً من أساليب التكرار (حلقات) التي تستخدم لتكرار أجزاء من البرنامج قدر ما تدعو الحاجة، لتحديد عدد مرات تكرار الحلقة تفحص كل حلقات **C++** ما إذا كان تعبير ما يساوى صحيح (true) أو خطأ (false) يبلغها هذا ما إذا كان عليها التكرار مرة إضافية أخرى أو التوقف فوراً.

هنالك ثلاثة أنواع من الحلقات في **C+ +**:-

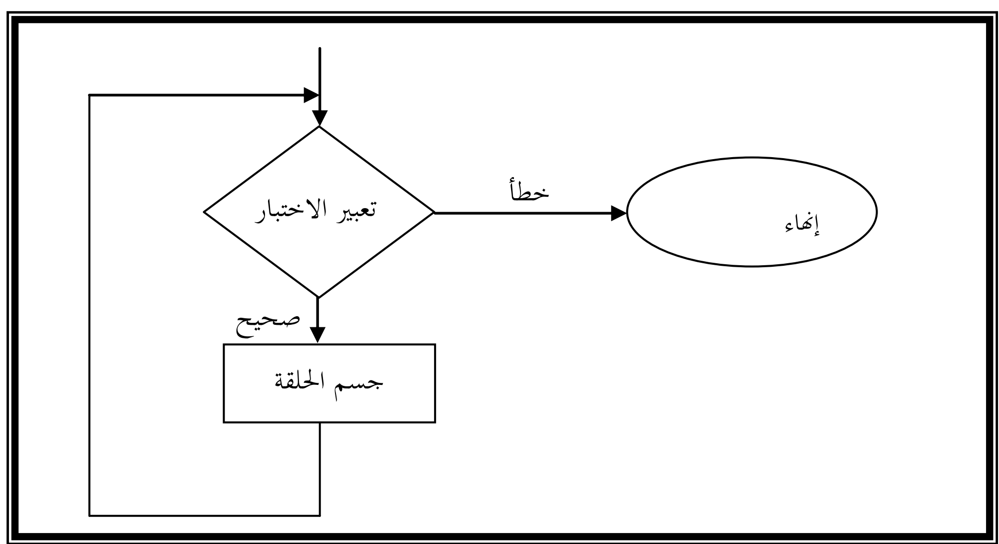
3.4.2 الحلقة while

تتيح الحلقة **while** تكرار فعل جزء من البرنامج إلى أن يتغير شرط ما .

فمثلاً:-

```
while (n<100)
n=n*2
```

ستستمر هذه الحلقة في مضاعفة المتغير **n** إلى أن تصبح قيمة **n** أكبر من 100 عندها تتوقف . تتكون الحلقة من الكلمة الأساسية **while** يليها تعبير اختبار بين أقواس ويكون جسم الحلقة محصوراً بين أقواس حاصرة { } إلا إذا كان يتألف من عبارة واحدة.
الشكل (1-3) يبين طريقة عمل الحلقة **while**:-



شكل (1-3) – طريقة عمل الحلقة while

مما يجدر التنويه إليه هنا أنه يتم فحص تعبير الاختبار قبل تنفيذ جسم الحلقة، وعليه لن يتم تنفيذ جسم الحلقة أبداً إذا كان الشرط خطأ عند دخول الحلقة وعليه المتغير **n** في المثال السابق يجب تمهيده عند قيمة أقل من 100 .

مثال :

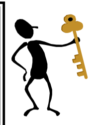
```
//Program 3-3:
#include<iostream.h>
main ( )
{
int counter, grade, total ,average;
total = 0;
counter = 1;
while (counter <= 0)   {
cout<< " Enter grade :  ";
cin >>grade;
total = total + grade;
```

```
counter = counter + 1;
}
cout<<endl;
average = total /10;
//Continued
cout << " Class average is: " << average <<endl;
return 0;
```

الخرج من البرنامج:



```
 Enter grade: 75   65  50  89  71  54 86  79 81 90
Class average is : 74
```



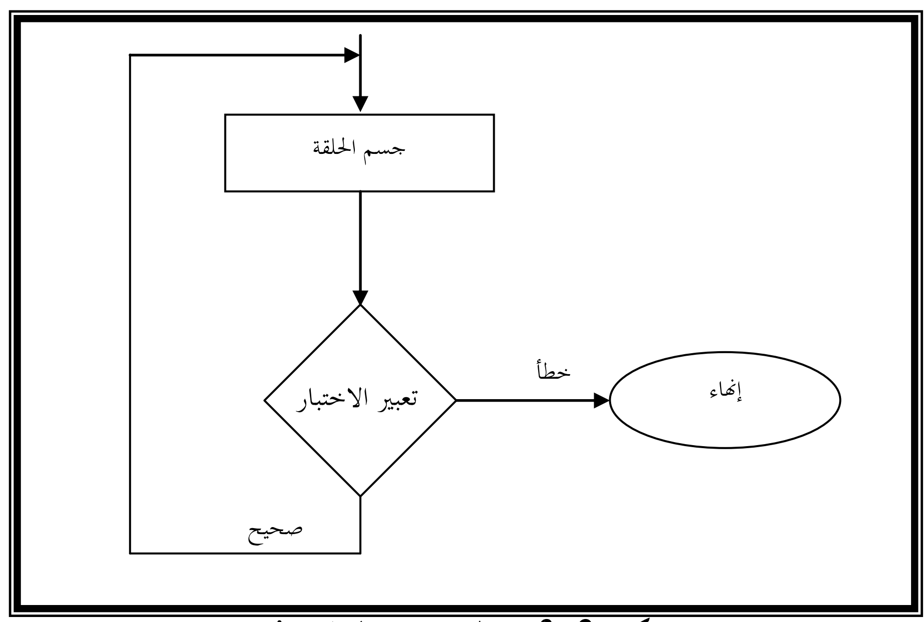
ما هو الخطأ في الحلقة الآتية:

```
while(c<5) {
   product *=c;
   ++c;
```

3.4.3 الحلقة do

تعمل الحلقة do (غالباً تسمى ...do...while) كالحلقة while، إلا أنها تفحص تعبير الاختبار بعد تنفيذ جسم الحلقة. وتستخدم أيضاً عندما نريد القيام بجزء من البرنامج مرة واحدة على الأقل.

الشكل(2-5) يبين كيفية عمل الحلقة do.



شكل (2-3) – طريقة عمل الحلقة do

تبدأ الحلقة do بالكلمة الأساسية do يليها جسم الحلقة بين أقواس حاصرة } { ثم الكلمة الأساسية while ثم تعبير اختبار بين أقواس ثم فاصلة منقوطة.

مثال:-

البرنامج التالي يقوم بطباعة الأعداد من 1 إلى 10 .

```
//Program 3-4:
// using do repetition structure
#include<iostream.h>
 main  (  )
{ int counter = 1;
do
cout << counter <<''   '' ;
```

```
while (+ + counter <= 10);
//Continued
return 0;
}
```

تقوم ;"   " >>cout بطباعة مسافة خالية بين كل رقم والآخر وعليه الخرج من البرنامج يكون كالتالي:

```
1 2 3 4 5 6 7 8 9 10
```

3.4.5 الحلقة for

عادة لا تعرف الحلقات do و while عدد مرات تكرار الحلقة. لكن في الحلقة for يكون عدد مرات تنفيذ الحلقة مذكوراً عادة في بدايتها.

المثال التالي يقوم بطباعة قيم المتغير counter من 1 إلى 10 .

```
//Program 3-5:
// using the for repetition structure
#include<iostream.h>
main( )
{
for ( int counter = 1; counter<= 10; counter++)
cout << counter <<endl ;
return 0;
}
```

الخرج من البرنامج

```
1
2
3
4
5
6
7
8
9
10
```

تحتوى الأقواس التي تلي الكلمة الأساسية for على ثلاثة تعابير مختلفة تفصلها فاصلة منقوطة. تعمل هذه التعابير الثلاثة في أغلب الأوقات على متغير يدعى متغير الحلقة ، وهو المتغير counter في المثال السابق.

هذه التعابير هي:-

تعبير التمهيد، الذي يمهد قيمة متغير الحلقة عادة ;int counter = 1 .

تعبير الاختبار، الذي يفحص عادة قيمة متغير الحلقة ليرى ما إذا كان يجب تكرار الحلقة مرة أخرى أو إيقافها ;counter <= 10.

تعبير التزايد، الذي يقوم عادة بزيادة (أو إنقاص) قيمة متغير الحلقة ++counter .

المثال التالي يقوم بإنقاص متغير الحلقة بـ 1 كلما تكررت الحلقة

```
//Program 3-6:
#include <iostream.h>
main ( )
{
          for ( int j=10; j>0; -- j)
          cout <<j<<'   ';
          return 0;
{
```

ستعرض هذه الحلقة

```
1 2 3 4 5 6 7 8 9 10
```

ويمكن أيضاً زيادة أو إنقاص متغير الحلقة بقيمة أخرى .

البرنامج التالي يوضح ذلك :

```
//Program 3-7:
#include<iostream.h>
main ( )
{
```

```
for (int j=10; j<100; j+=10)
cout <<j<<'    ';
 return 0;
{
```

ستعرض :-

```
10  20  30  40  50  60  70  80  90  100
```

يمكن استعمال عدة عبارات في تعبير التمهيد وتعبير الاختبار كما في البرنامج التالي :-

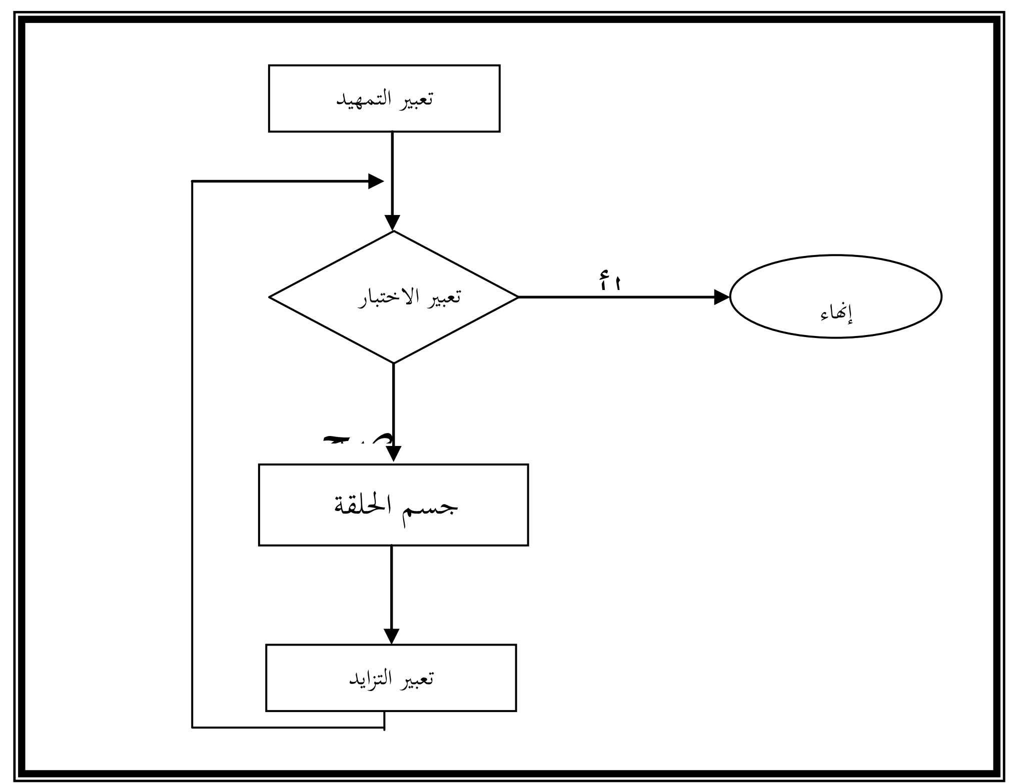
```
//Program 3-8:
#include<iostream.h>
main ( )
{
for ( int j=0;int total=0; j<10; ++ j;total+=j)
 cout <<total<<'    ';
return 0 ;
}
```

تعرض:-

```
0  1  3  6  10  15  21  28  36  45
```

أيضاً يمكن في الحلقة for تجاهل أحد التعابير أو ثلاثتها كلياً مع المحافظة على الفواصل المنقوطة فقط.

الشكل(2-6) يبين كيفية عمل الحلقة for.



شكل (3-3) – طريقة عمل الحلقة for

3.4.6 الحلقات المتداخلة

تأخذ الحلقات for المتداخلة الشكل العام التالي :-

```
for (..........)
            for (..........)
                      for (..........)

                              statements;
```

مثال:

```
//Program 3-9:
// An Example of 2 nested loops
#include<iostream.h>
main( )
{
int i,j;
for (i=1 ; i<3;++i)
{
for (j=1 ; j<4;++j)
cout << i*j<<'  ' <<endl;
}
return 0;
}
```

نلاحظ هنا أن الحلقة الداخلية تتكرر 4 مرات لكل قيمة من قيم **i** (عداد الحلقة الخارجية).

الخرج من البرنامج:

```
1  2  3  4
2  4  6  8
   6  9  12
```

يمكننا وضع أي نوع من الحلقات ضمن أي نوع آخر، ويمكن مداخلة الحلقات في حلقات متداخلة في حلقات أخرى وهكذا.

3.4.7 التحكم بالحلقات

تعمل الحلقات عادة بشكل جيد إلا أننا في بعض الأوقات نحتاج للتحكم بعمل الحلقات ، العبارتين break وcontinue توفران هذه المرونة المطلوبة.

العبارة break :-

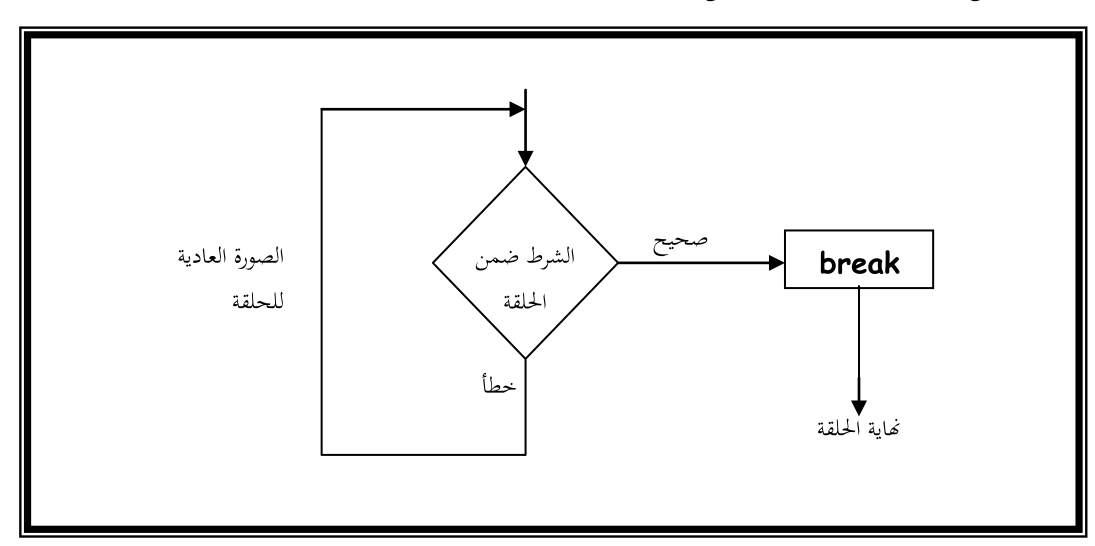
تتيح لنا العبارة break الخروج من الحلقة في أي وقت.

المثال التالي يبين لنا كيفية عمل العبارة break :

```
//Program 3-10:
//An Example on break as a loop exit
#include<iostream.h>
main( )
{
int isprime ,j ,n;
isprime = 1;
cin>>n;
for (j=2,j<n;++j)
{
if (n%j== 0)
{
isprime =0;
break;
}
}
}
```

هذا البرنامج يجعل قيمة المتغير **isprime** عند 1 إذا كان n عدد أولى prime يجعل قيمته0 إذا لم يكن كذلك (الرقم الأولي هو رقم يقبل القسمة على نفسه وعلى الرقم واحد فقط).لمعرفة ما إذا كان الرقم أولياً أم لا تتم قسمته على كل الأرقام وصولاً إلى n-1 ، إذا قبل الرقم n القسمة على أحد هذه القيم من دون باقي فإنه لا يكون أولياً لكن إذا قبل الرقم n القسمة على أحد هذه القيم بشكل صحيح لا داعي لإكمال الحلقة فحالما يجد البرنامج الرقم الأول الذي يقسم n بشكل صحيح يجب أن يضبط قيمة المتغير isprime عند 0 ويخرج فوراً من الحلقة.

الشكل(3-4) يبين طريقة عمل العبارة **break**:-



شكل (3-4) – طريقة عمل العبارة break

العبارة continue :-

تعيد العبارة continue التنفيذ إلى أعلى الحلقة.

المثال التالي يوضح كيفية عمل العبارة **continue**:-

```
//Program 3-11:
//An Example on continue statement
#include<iostream.h>
main( )
{
int dividend , divisor;
```

```
do
//Continued
{
cout << ''Enter dividend:    '';
cin>>dividend;
cout<< ''Enter divisor:    '';
//Continued
cin>>divisor;
if( divisor == 0)
{
cout<<" divisor can't be zero\n" ;
continue;
}
cout <<"Quotient is "<< dividend/divisor;
cout<<" do another (y/n)?";
cin>>ch
}
while (ch! = 'n');
}
```

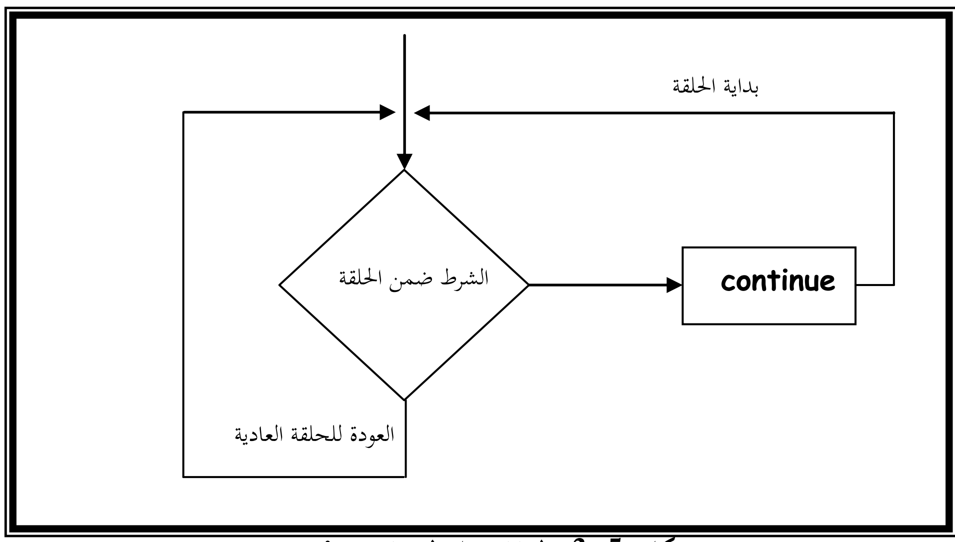
القسمة على 0 أمر غير مرغوب فيه لذا إذاكتب المستخدم 0 على أنه القاسم يعود التنفيذ إلى أعلى الحلقة ويطلب من البرنامج إدخال قاسم ومقسوم جديدين.

```
If ( divisor == 0)
{
cout << "divisor can't be zero\n";
continue;
}
```

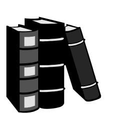
يستمر تنفيذ الحلقة إلى أن يدخل المستخدم الحرف **n** .

```
while( ch ! ='n' ) ;
```

الشكل (3-5) يبين طريقة عمل العبارة **continue**.



شكل (3-5) طريقة عمل العبارة continue



الملخص:

- توفر لغة C++ عوامل تسمى عوامل التعيين الحسابي وهي =+ ، = - ، =*، =/ و =%.
- توفر C++ عاملي التزايد ++ والتناقص -- واللذين يقومان بزيادة وإنقاص قيمة متغير ما بمقدار 1 .
- تأخذ الحلقة for الشكل العام التالي:

```
for( expression1; expression2; expression3)
statement
```

حيث يمثل:

expression1 تعبير التمهيد الذي يمهد قيمة متغير الحلقة.

expression2 تعبير الاختبار الذي يفحص قيمة متغير الحلقة ويحدد ما إذا كان يجب تكرار الحلقة مرة أخرى أم لا.

expression3 يمثل تعبير التزايد الذي يقوم بزيادة أو إنقاص قيمة متغير الحلقة.

- تأخذ الحلقة while الشكل العام التالي:

```
while( condition)
statement
```

- تأخذ الحلقة do الشكل التالي :

```
do
statement
while( condition)
```

- الحلقة do تفحص تعبير الاختبار بعد تنفيذ جسم الحلقة ، وعليه يتم تكرار جسم الحلقة do مرة واحدة على الأقل.
- تستعمل العبارة break للخروج من الحلقة في أي وقت.
- تعيد العبارة continue التنفيذ إلى بداية الحلقة.
- تستعمل العبارة switch للاختيار بين عدة خيارات مختلفة بناءاً على قيمة متغير ما.
- تستعمل العوامل المنطقية لكتابة تعابير مركبة وهي &&، || و ! والتي تعني and، or و not على التوالي.



الأسئلة

1/استعمل العبارات في السؤال الأول من الوحدة السابقة لكتابة برنامج **C++** يقوم برفع المتغير **x** للأس **y** باستخدام الحلقة **while**.

2/ ما هو الخطأ في الآتي:

```
cin << value;
```

3/ ما هو الخطأ في الحلقة **while** التالية:-

```
while (z>= 0)
       sum += z;
```

4/أكتب برنامجاً يستقبل عدد من لوحة المفاتيح ثم يحدد ما إذا كان الرقم زوجياً أم فردياً. (تلميح: استخدم العامل **(%)**).

5/ ما هي مخرجات البرنامج التالي:

```
#include <iostream.h>
main ( )
{
int y, x = 1, total =0;
while (x<= 10)  {
y = x+x;
cout <<y << endl;
total +=y;
++x;
}
cout << " total is: " << total << endl;
return 0;
}
```

6/ مضروب العدد الموجب **n** يعرف كالآتي:

$$n! = n.\,(n-1).\,(n-2)\;....\;1$$

أكتب برنامج **C++** يقوم باستقبال رقم من المستخدم. ويقوم بحساب وطباعة مضروبه.

7/ أوجد الخطأ في الجزء التالي:

```
for ( x = 100, x >= 1, x++)
```

```
        cout << x << endl;
```

❑ الجزء التالي يقوم بطباعة الأعداد الزوجية من **19** إلى **1**

```
for ( x = 19 ; x >= 1 , x+=2)
        cout << x << endl;
```

8/ ما هو الغرض من البرنامج التالي:

```
#include <iostream.h>
main ( )
{
int  x  ,y ;
cout << "Enter two integers in the range 1-20";
cin >> x>> y;
for (int I = 1; I < = y ; I++)  {
     for ( int j = 1; j <= x;  j++)
        cout << " ";
        cout << endl;
        }
return 0;
}
```

4.0

الوحدة الرابعة

الدوال Functions



بنهاية هذه الوحدة :

- ستتمكن من تقسيم برنامجك إلى أجزاء صغيرة تسمى دوال (Functions) .
- ستتعرف على أغلب الدوال الرياضية الجاهزة والمعرفة في الملف math.h والتي تقوم بالعمليات الرياضية.
- ستتعرف على كيفية كتابة الدوال في C++ .

14. المقدمة

ورثت اللغة ++C من اللغة C مكتبة ضخمة وغنية بدوال تقوم بتنفيذ العمليات الرياضية، التعامل مع السلاسل والأحرف، الإدخال والإخراج، اكتشاف الأخطاء والعديد من العمليات الأخرى المفيدة مما يسهل مهمة المبرمج الذي يجد في هذه الدوال معيناً كبيراً له في عملية البرمجة.

يمكن للمبرمج كتابة دوال تقوم بأداء عمليات يحتاج لها في برامجه وتسمى مثل هذه الدوال

Programmer- defined functions

24. فوائد استخدام الدوال في البرمجة

1/ تساعد الدوال المخزنة في ذاكرة الحاسب على اختصار البرنامج إذ يكتفي باستدعائها باسمها فقط لتقوم بالعمل المطلوب .

2/ تساعد البرامج المخزنة في ذاكرة الحاسب أو التي يكتبها المستخدم على تلافى عمليات التكرار في خطوات البرنامج التي تتطلب عملاً مشابهاً لعمل تلك الدوال.

3/ تساعد الدوال الجاهزة في تسهيل عملية البرمجة.

4/ يوفر استعمال الدوال من المساحات المستخدمة في الذاكرة.

5/ كتابة برنامج الـ + +C في شكل دوال واضحة المعالم يجعل البرنامج واضحاً لكل من المبرمج والقارئ على حد سواء.

34. مكتبة الدوال الرياضية (Math Library Functions)

تحتوى مكتبة الدوال الرياضية على العديد من الدوال التي تستخدم في تنفيذ العمليات الرياضية الحسابية. فمثلاً المبرمج الذي يرغب في حساب وطباعة الجذر التربيعي للعدد 900 قد يكتب عبارة كالتالية:

```
cout << sqrt ( 900);
```

عند تنفيذ هذه العبارة يتم استدعاء الدالة المكتبية **sqrt** لحساب الجذر التربيعي للعدد بين القوسين (900). يسمى العدد بين القوسين وسيطة الدالة **argument** وعليه فالعبارة السابقة تقوم بطباعة العدد 30 ، تأخذ الدالة **sqrt** وسيطة من النوع **double** وتكون النتيجة قيمة من نفس النوع وينطبق هذا على جميع الدوال الرياضية.

عند استعمال الدوال الرياضية في أي برنامج بلغة *C++* يجب تضمين الملف **math.h** والذي يحتوى على هذه الدوال.

الجدول التالي يلخص بعض الدوال الرياضية:

Function	Description	Example
sqrt(x)	الجذر التربيعي لـ x	sqrt (9.0) is 3
exp(x)	exp(1.0) is 2.718282	e^x
fabs(x)	القيمة المطلقة لـ x	if x > 0 fabs(x) = x = 0 fabs(x) = 0 < 0 fabs(x) = -x
ceil(x)	تقرب x لأصغر عدد صحيح أكبر من x	ceil(9.2) is 10.0 ceil(-9.8) is 9.0
floor(x)	تقرب x لأكبر عدد صحيح أصغر من x	floor(9.2) is 9 floor(-9.8) is -10.0

.44 الدوال المعرفة بواسطة المستخدم
Programmer-defined Functions

الدوال تمكن المبرمج من تقسيم البرنامج إلى وحدات **modules**، كل دالة في البرنامج تمثل وحدة قائمة بذاتها، ولذا نجد أن المتغيرات المعرفة في الدالة تكون متغيرات محلية **(Local)** ونعني بذلك أن المتغيرات تكون معروفة فقط داخل الدالة.

أغلب الدوال تمتلك لائحة من الوسائط **(Parameters)** والتي هي أيضاً متغيرات محلية.

هنالك عدة أسباب دعت إلى تقسيم البرنامج إلى دالات وتسمى هذه العملية **(Functionalizing a program)** وهي:

1/ تساعد الدوال المخزنة في ذاكرة الحاسب على اختصار البرنامج إذ يكتفي باستدعائها باسمها فقط لتقوم بالعمل المطلوب .

2/ تساعد البرامج المخزنة في ذاكرة الحاسب أو التي يكتبها المستخدم على تلافي عمليات التكرار في خطوات البرنامج التي تتطلب عملاً مشابهاً لعمل تلك الدوال.

3/ تساعد الدوال الجاهزة في تسهيل عملية البرمجة.

4/ يوفر استعمال الدوال من المساحات المستخدمة في الذاكرة.

5/ كتابة برنامج **+ C+** في شكل دوال واضحة المعالم يجعل البرنامج واضحاً لكل من المبرمج والقارئ على حد سواء.

كل البرامج التي رأيناها حتى الآن تحتوى على الدالة **main** وهي التي تنادى الدوال المكتبية لتنفيذ مهامها. سنرى الآن كيف يستطيع المبرمج بلغة الـ **C++** كتابة دوال خاصة به.

.54 نموذج الدالة
Function Prototype

عندما يولد المصرف تعليمات لاستدعاء دالة، ما فإنه يحتاج إلى معرفة اسم الدالة وعدد وسيطاتها وأنواعها ونوع قيمة الإعادة ، لذا علينا كتابة نموذج أو (تصريح) للدالة قبل إجراء أي استدعاء لها وتصريح الدالة هو سطر واحد يبلغ المصرف عن اسم الدالة وعدد وسيطاتها وأنواعها ونوع القيمة المعادة بواسطة الدالة. يشبه تصريح الدالة ، السطر الأول في تعريف الدالة ، لكن تليه فاصلة منقوطة.

فمثلا في تصريح الدالة التالي:-

```
int anyfunc(int);
```

النوع **int** بين القوسين يخبر المصرف بأن الوسيط الذي سيتم تمريره إلى الدالة سيكون من النوع **int** و **int** التي تسبق اسم الدالة تشير إلى نوع القيمة المعادة بواسطة الدالة.

4.6 تعريف الدالة Function Definition

يأخذ تعريف الدوال في *C++* الشكل العام التالي:

```
return-value-type   function-name (parameter list)
      {
            declarations and statements
      }
```

حيث:

return-value-type: نوع القيمة المعادة بواسطة الدالة والذي يمكن أن يكون أي نوع من أنواع بيانات *C++*. وإذا كانت الدالة لا ترجع أي قيمة يكون نوع إعادتها **void**.

function-name: اسم الدالة والذي يتبع في تسميته قواعد تسمية المعرفات (identefiers).

parameter list: هي لائحة الوسيطات الممرة إلى الدالة وهي يمكن أن تكون خالية (void) أو تحتوى على وسيطة واحدة أو عدة وسائط تفصل بينها فاصلة ويجب ذكر كل وسيطة على حدة.

declarations and statements: تمثل جسم الدالة والذي يطلق عليه في بعض الأحيان **block** . يمكن أن يحتوى الـ **block** على إعلانات المتغيرات ولكن تحت أي ظرف لا يمكن أن يتم تعريف دالة داخل جسم دالة أخرى. السطر الأول في تعريف الدالة يدعى المصرح **declarator** والذي يحدد اسم الدالة ونوع البيانات التي تعيدها الدالة وأسماء وأنواع وسيطاتها.

4.7 إستدعاء الدالة Function Call

يتم استدعاء الدالة (التسبب بتنفيذها من جزء آخر من البرنامج، العبارة التي تفعل ذلك هي استدعاء الدالة) يؤدى استدعاء الدالة إلي انتقال التنفيذ إلي بداية الدالة.

يمكن تمرير بعض الوسيطات إلى الدالة عند استدعائها وبعد تنفيذ الدالة يعود التنفيذ للعبارة التي تلي استدعاء الدالة.

4.8 قيم الإعادة Returned Values

بإمكان الدالة أن تعيد قيم إلي العبارة التي استدعتها. ويجب أن يسبق اسم الدالة في معرفها وإذا كانت الدالة لا تعيد شيئاً يجب استعمال الكلمة الأساسية **void** كنوع إعادة لها للإشارة إلي ذلك .

هنالك ثلاث طرق يمكن بها إرجاع التحكم إلى النقطة التي تم فيها استدعاء الدالة:

١ / إذا كانت الدالة لا ترجع قيمة يرجع التحكم تلقائياً عند الوصول إلى نهاية الدالة

٢ / باستخدام العبارة **return;**

٣ / إذا كانت الدالة ترجع قيمة فالعبارة **return expression;** تقوم بإرجاع قيمة التعبير **expression** إلى النقطة التي استدعتها .

خذ برنامجاً يستخدم دالة تدعى **square** لحساب مربعات الأعداد من 1 إلى 10.

مثال:

```
//Program 4-1:
#include<iostream.h>
int square(int);//function prototype
main()
{
      for(int x=1;x<=10;x++)
  cout<<square(x)<<" ";
   cout<<endl;
}
//now function definition
int square(int y)
{
      return y*y;
}
```

الخرج من البرنامج يكون كالآتي:

1	4	9	16	25	36	49	64	81	100

يتم استدعاء الدالة **square** داخل الدالة **main** وذلك بكتابة **square(x)**.
تقوم الدالة **square** بنسخ قيمة **x** في الوسيط **y** . ثم تقوم بحساب **y*y** ويتم إرجاع النتيجة إلى الدالة **main** مكان استدعاء الدالة **square**، حيث يتم عرض النتيجة وتتكرر هذه العملية عشر مرات باستخدام حلقة التكرار **for** .

تعريف الدالة **square( )** يدل على أنها تتوقع وسيطة من النوع **int** . و **int** التي تسبق اسم الدالة تدل على أن القيمة المعادة من الدالة **square** هي من النوع **int** أيضاً .
العبارة **return** تقوم بإرجاع ناتج الدالة إلى الدالة **main**.
السطر:

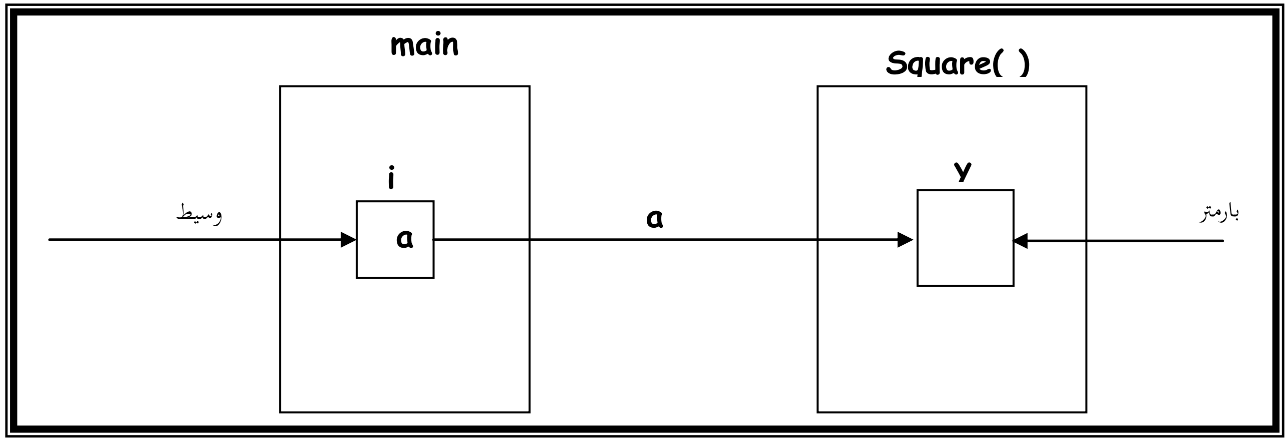
```
int square (int)
```

هو نموذج أو تصريح الدالة **(function prototype)** .

.94 الوسيطات Parameters

الوسيطات هي الآلية المستخدمة لتمرير المعلومات من استدعاء الدالة إلي الدالة نفسها حيث يتم نسخ البيانات وتخزن القيم في متغيرات منفصلة في الدالة تتم تسمية هذه المتغيرات في تعريف الدالة.فمثلاً في المثال السابق تؤدى العبارة **cout<< square(a);** في **main( )** إلى نسخ القيمة **a** إلى البارمتر **y** المعرف في تعريف الدالة.

المصطلح وسيطات **Argument** يعنى القيم المحددة في استدعاء الدالة بينما يعنى المصطلح بارمترات **parameters** المتغيرات في تعريف الدالة والتي تم نسخ تلك القيم إليها، ولكن غالباً ما يتم استعمال المصطلح وسيطات لقصد المعنيين.
الشكل (1-4) يوضح هذا.



شكل (4-1) يوضح كيفية تمرير الوسائط .

البرنامج التالي يستخدم دالة تدعى **maximum** والتي ترجع العدد الأكبر بين ثلاثة أعداد صحيحة.

يتم تمرير الأعداد كوسائط للدالة التي تحدد الأكبر بينها وترجعه للدالة **main** باستخدام العبارة **return** ويتم تعيين القيمة التي تمت إعادتها إلى المتغير **largest** الذي تتم طباعته.

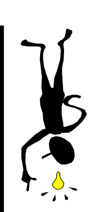
```
//Program 4-2:
#include <iostream.h>
int maximum (int, int, int);
main(   )
{
int a, b, c;
cout << "Enter three integers: " ;
cin >> a >> b >> c ;
cout << " maximum is : " << maximum (a, b, c) << endl;
return 0;
}
int maximum (int x, int y, int z)
{
int max = x;
if (y > x)
max = y;
```

```
if (z > max)
max = z;
//Continued
return max;
}
```

الخرج من البرنامج بافتراض أن المستخدم قد أدخل الأرقام 22، 85، 17.



Enter three integers: **22** **85** **17**
Maximum is: 85

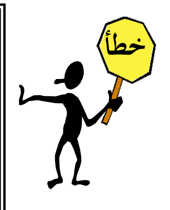


- رغم أنها غير ضرورية إلا أنه يتم ذكر أسماء الوسيطات في التصريح ومن غير الضروري أن تكون هذه الأسماء هي نفسها المستعملة في تعريف الدالة . في الواقع، المصرف يتجاهلها لكنها تكون مفيدة أحياناً للذين يقرأون البرنامج . فمثلاً لنفترض أن الوسيطين x و y تمثلان إحداثيات الشاشة في دالة تفرض نقطة على الشاشة.
- void draw_dot (int,int); [تصريح الدالة]

هذا التصريح كافي للمعرف لكن المبرمج قد لا يعرف أيهما الإحداثي السيني وأيهما الإحداثي الصادي. لذا سيكون مفيداً لو كتبنا :

```
void draw_dot (int x,int y);
```

- إذا لم يذكر المبرمج نوع إعادة الدالة في تصريح الدالة يفترض المصرف أن نوع الدالة هو int.



- عدم كتابة نوع إعادة الدالة في تعريف الدالة إذا كان الإعلان عن الدالة يتطلب نوع إعادة غير int.
- إرجاع قيمة من دالة تم الإعلان عن نوع إعادتها void.

4.10 دوال بدون وسيطات Functions with Empty Pararmeter Lists

في لغة الC++ تكتب الدالة التي لا تمتلك وسيطات إما بكتابة void بين القوسين الذين يتبعان اسم الدالة أو تركهما فارغين ، فمثلاً الإعلان

```
void print ( );
```

يشير إلى أن الدالة print لا تأخذ أي وسيطات وهي لا ترجع قيمة .

المثال التالي يبين الطريقتين اللتين تكتب بهما الدوال التي لا تأخذ وسيطات:

```
//Program 4-3:
// Functions that take no arguments
#include <iostream.h>
void f1 ( );
```

```
void f2 (void);
//Continued
main( )
{
    f1 ( );
    f2 ( );
return 0;
}

void f1 ( )
{
    cout << "Function f1 takes no arguments" << endl;
}
void f2 (void)
{
    cout << "Function f2 also  takes no arguments" <<
endl;
}
```

الخرج من البرنامج:

```
Function f1 takes no arguments
Function f2 also  takes no arguments
```

.114 الدوال السياقية Inline Functions

تحتاج بعض التطبيقات التي يعتبر فيها وقت تنفيذ البرنامج أمراً حيوياً وحاسماً، لإبدال عملية استدعاء واستخدام دالة بما يسمى دالة سياقية . وهى عبارة عن شفرة تقوم بالعمل المطلوب نفسه، يتم تعريف الدالة السياقية باستعمال نفس التركيب النحوي المستخدم لتعريف الدالة الاعتيادية ، لكن بدلاً من وضع شفرة الدالة في مكان مستقل يضعها المصرف في

السياق الطبيعي للبرنامج مكان ظهور استدعاء الدالة. يتم جعل الدالة سياقية عن طريق استخدام الكلمة الأساسية **inline** في تعريف الدالة.

```
inline void func1( )
{
statements
}
```

تستخدم الدالة السياقية فقط إذا كانت الدالة قصيرة وتستخدم مرات عديدة في البرنامج.

مثال:

```
//Program 4-4:
#include<iostream.h>
inline float cube(float s){return s*s*s;}
main()
{
  cout<<"\nEnter the side length of your cube : ";
      float side;
  cin>>side;
  cout<<"volume of cube is "
  <<cube(side)
  <<endl;
}
```

الخرج من البرنامج:

```
Enter the side length of your cube : 5
volume of cube is 125
```

مثال آخر على الدوال السياقية :

```
// Program 4-5:
```

```
#include <iostream.h>
inline int mult( int a, int b)
{
return (a*b);
}
//Continued
main( )
{
int x, y, z;
cin >> x >> y >> z;
cout << "x = " << x << " y = " << y << " z = " << z << endl;
cout << "product1" << mult (x ,y) << endl;
cout << "product2" << mult (x +2, y) << endl;
return 0;
}
```

الخرج من البرنامج إذا أدخلنا (x = 3, y =4, z = 5):



```
x = 3    y =4    z = 5
product1   12
product2    32
```

.124 تحميل الدالات بشكل زائد Overloading Functions

تحميل الدالات بشكل زائد يعنى استعمال الاسم لعدة دالات لكن كل دالة يجب أن يكون لها تعريف مستقل. عند استدعاء دالة يبحث المصرف عن نوع وسيطات الدالة وعددها لمعرفة الدالة المقصودة . ولكي يميز المصرف بين دالة وأخرى تحمل نفس الاسم ، يقوم بعملية تعرف بتشويه الأسماء (names mangling)، تتألف هذه العملية من إنشاء اسم جديد خاص بالمصرف عن طريق دمج اسم الدالة مع أنواع وسيطاتها.

مثال:

البرنامج التالي يقوم بتحميل الدالة **square** بشكل زائد لحساب الجذر التربيعي للنوع **int** وللنوع **double** :-

```
//Program 4-6:
#include <iostream.h>
int square(int x){return x*x;}
//Continued
double square(double y){return y*y;}
main ()
{
cout<< " The square of integer 7 is"
<<" "<<square(7)<< endl
<<"The square of double 7.5 is"
<<" "<<square(7.5)<< endl;
return 0;
}
```

الخرج من البرنامج:

```
The square of integer 7 is 49
The square of double 7.5 is 56.25
```

إليك الآن برنامجاً يقوم بتحميل دالة تدعى **abs** لحساب القيمة المطلقة لأعداد من النوع **int** ، **double** و **long**.

```
//Program 4-7:
#include <iostream.h>
// abs is overloaded three ways
int abs (int i);
double abs(double d);
long abs(long l);
```

```
int main( )
{
cout<< abs (-10)<<"\n";
cout<< abs (-11.0)<<"\n";
cout<< abs (-9L)<<"\n";
return 0;
}
int abs (int i)
//Continued
{
cout<<"using integer abs( )\n";
return i<0 ? -i :i ;
}
double abs (double d)
{
cout<<" using double abs( )\n";
return d<0.0 ? -d : d ;
}
long abs(long l)
{
cout<<" using long abs( )\n";
return l<0.0 ? -l : l ;
}
```

الخرج من البرنامج:

```
using integer abs( )
10
using double abs( )
11.0
using long abs( )
9L
```

الوسيطات الافتراضية
Default Arguments
.134

تتيح الوسيطات الافتراضية تجاهل وسيطة أو أكثر عند استدعاء الدالة ، وعند وجود وسيطة ناقصة يزود تصريح الدالة قيماً ثابتة لتلك الوسيطات المفقودة .

مثال:-

```
//Program 4-8:
#include <iostream.h>
inline box_volume (int length=1,int width=1,int height=1)
{return length*width*height;}
main()
{
cout<<"The default box volume is "
<<box_volume() <<endl
<<"width 1 and height 1 is "
<<box_volume(10)<<endl;
return 0;
}
```

الخرج من البرنامج:

```
The default box volume is 1
Width 1 and height1 is 10
```

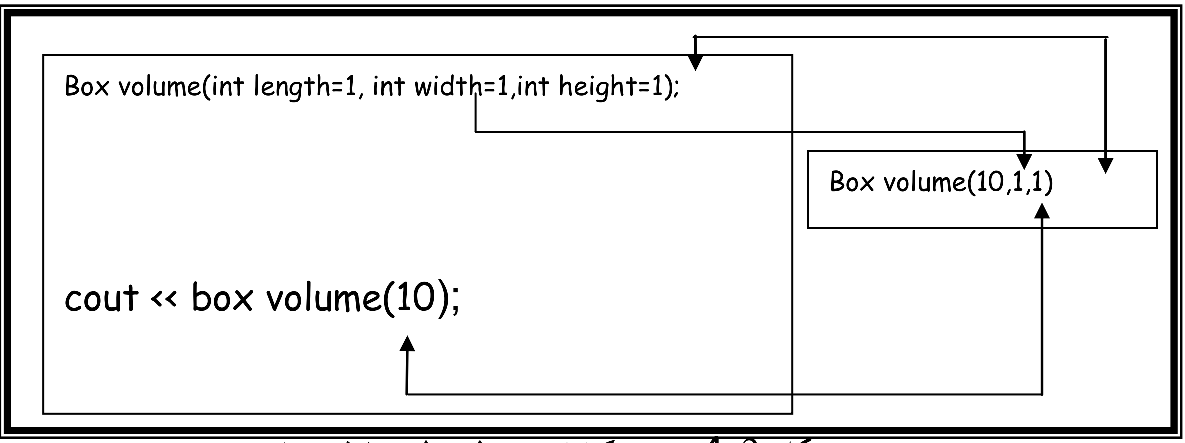
تم استعمال تصريح الدالة لتزويد الدالة box_volume بثلاث وسيطات افتراضية وتحدد القيمة التي تلي علامة المساواة قيمة هذه الوسيطات وهى 1 لكل وسيطة .

يستدعى البرنامج في main الدالة box_volume بطريقتين:-

أولاً: بدون وسيطات لذا تم احتساب box_volume باستخدام القيم الافتراضية للوسيطات لتعيد الدالة القيمة 1 كحجم للمربع.

ثانياً : بوسيطة واحدة وهى 10 لتعيد الدالة 10 حجم للمربع ، في هذه الحالة length = 10.

الشكل (4-2) يبين كيف يزود تعريف الدالة الوسيطات الافتراضية:



شكل (4-2) يوضح كيفية تزويد الوسيطات الإفتراضية

فقط الوسيطات الموجودة في نهاية لائحة الوسيطات يمكن إعطاؤها وسيطات افتراضية ،فإذا كانت هنالك وسيطة واحدة فقط لها وسيطة افتراضية يجب أن تكون الأخيرة ولا يمكننا وضع وسيطة افتراضية في وسط لائحة وسيطات عادية بمعنى آخر لا يمكننا كتابة

```
int box_volume(int length, int width=1,int height);
```

لأن الوسيطة الافتراضية ليست الوسيطة الأخيرة.

144. التمرير بالقيمة والتمرير بالمرجع

لنفرض أننا لدينا متغيرين صحيحين في برنامج ونريد استدعاء دالة تقوم بتبديل قيمتي الرقمين ،لنفرض أننا عرفنا الرقمين كالآتي:

```
int x=1;
int y=2;
```

أ/ التمرير بالقيمة(pass-by-value):-

ترى هل تقوم الدالة التالية بتبديل القيمتين:

```
void swap (int a, int b)
{
```

```
int temp =a;
a=b.
b=temp;
}
```

تقوم هذه الدالة بتبديل قيمتي a و b ، لكن إذا استدعينا هذه الدالة كالآتي:

```
swap( x,y);
```

سنجد أن قيمتي x و y لم تتغير وذلك لأن الوسيطات الاعتيادية للدالة يتم تمريرها بالقيمة وتنشئ الدالة متغيرات جديدة كلياً هي a و b في هذا المثال لتخزين القيم الممررة إليها وهي (1,2) ثم تعمل على تلك المتغيرات الجديدة وعليه عندما تنتهي الدالة ورغم أنها قامت بتغيير a إلى 2 و b إلى 1 لكن المتغيرات x و y في استدعاء الدالة لم تتغير.

ب/ التمرير بالمرجع(pass-by-refrence):

التمرير بالمرجع هو طريقة تمكن الدالة ( )swap من الوصول إلى المتغيرات الأصلية x و y والتعامل معها بدلاً من إنشاء متغيرات جديدة . ولإجبار تمرير الوسيطة بالمرجع نضيف الحرف & إلى نوع بيانات الوسيطة في تعريف الدالة وتصريح الدالة .

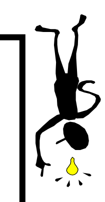
المثال (3-4) يبين كيفية كتابة الدالة swap وتمرير وسيطاتها بالمرجع:

```
//Program 4-9:
#include <iostream.h>
void swap (int & , int&);
main ( )
{
int x= 1;
int y= 2;
 swap (x, y);
return 0;
}
void swap (int& a, int & b)
{
cout <<"Original value of a is " << a<<endl;
int temp =a;
a=b;
b=temp;
```

```
cout <<"swapped value of a is " << a<<endl;
}
```

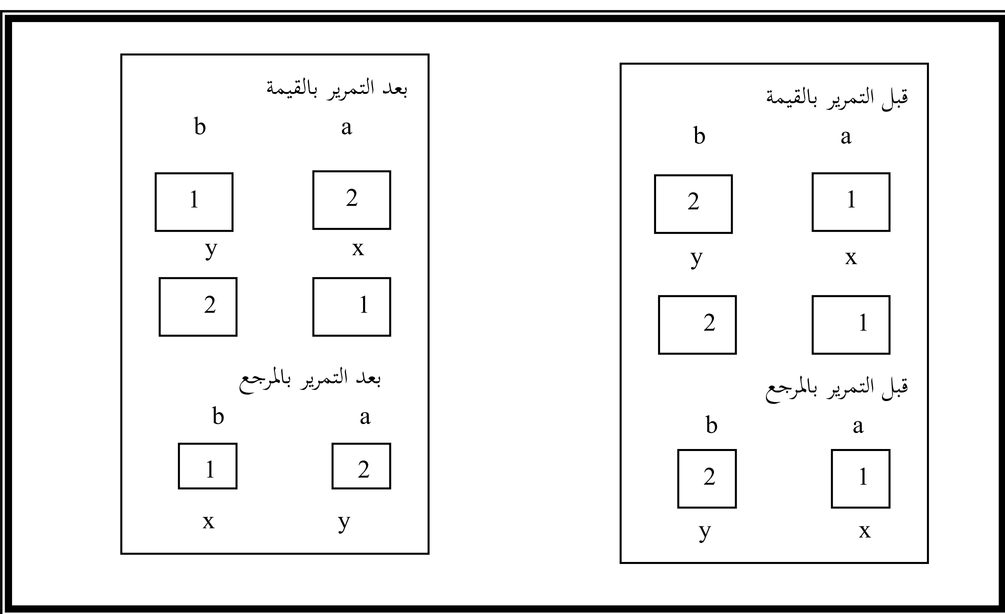
بعد تنفيذ هذه الدالة تتغير قيمة x إلى 2 و y إلى 1 . ويكون الخرج من البرنامج كالتالي:

```
Original value of a is 1
Swapped value of a is 2
```



الحرف & يلي int في التصريح والتعريف وهو يبلغ المصرف أن يمرر هذه الوسيطات بالمرجع، أي أن الوسيطة a هي مرجع إلى x و b هي مرجع إلى y ولا يستعمل & في استدعاء الدالة.

الشكل (3-4)يبين الفرق بين التمرير بالمرجع والتمرير بالقيمة.



شكل (3-4) يوضح طريقتي التمرير بالمرجع والتمرير بالقيمة.

الملخص:

- أفضل طريقة لتطوير وصيانة البرامج الكبيرة هو تقسيمها لوحدات صغيرة تسمى دوال.
- يتم تنفيذ الدوال عن طريق استدعائها .
- استدعاء الدالة يكون بكتابة اسم الدالة متبوعاً بوسيطاتها وأنواع تلك الوسائط.
- الصورة العامة لتعريف الدالة هو:-

```
return-value-type function-name( parameters-list)
{
    declarations and statements
}
```

حيث:-

type-value-return يمثل نوع البيانات الذي تعيده الدالة ، إذاكانت الدلالة لا تعيد قيمة يكون **void**.

function name يمثل اسم الدالة ويتبع في تسميته قواعد تسمية المتغيرات .

parameters_list هي لائحة من المتغيرات تفصلها فاصلة وتمثل الوسيطات التي سيتم تمريرها إلى الدالة.

- نموذج أو تصريح الدالة **(function prototype)** يمكن المصرف من معرفة ما إذا تم استدعاء الدالة بالصورة الصحيحة.
- يتجاهل المصرف أسماء المتغيرات المذكورة في تصريح الدالة.
- يمكن استعمال نفس الاسم لعدة دالات ، لكن يجب أن يكون لكل دالة تعريف مستقل ويسمى هذا بتحميل الدالات بشكل زائد **(function overloading)**.
- تسمح **C++** بتمرير وسيطات افتراضية وعليه عند تجاهل وسيطة أو أكثر في استدعاء الدالة يزود تصريح الدالة قيم تلك الوسيطات المفقودة.

الأسئلة

1/ أكتب تصريحاً **(prototype)** لدالة **smallest** والتي تأخذ ثلاث أعداد صحيحة **x** ، **y** و **z** كوسيطات لها وترجع قيمة من النوع **int**.

2/ أكتب تعريفاً لدالة ترجع الأكبر من بين ثلاثة أرقام صحيحة.

3/ أكتب تعريفاً لدالة تحدد ما إذا كان الرقم رقماً أولياً أم لا.

تلميح: الرقم الأولى هو الذي لا يقبل القسمة إلا على نفسه والرقم **1**.

4/ جد الخطأ في الدالة الآتية:

```
void product (      ) {
       int a, b, c, result;
       cout << " Enter three integers:  ";
       cin >> a>> b >>c;
       result = a*b*c;
       cout << "Result is : " << result;
       return result;
}
```

5/ جد الخطأ في الدالة الآتية:-

```
void  f(float  a);   {

       cout << a << endl;
       }
```

6/ أكتب تصريحاً لدالة تدعى **instructions** والتي لا تأخذ أي وسيطات ولا ترجع أي قيمة.

7/ أكتب تعريفاً لدالة تستقبل عدداً من المستخدم ثم ترجع العدد معكوساً فمثلاً إذا أدخل المستخدم العدد **1234** ترجع الدالة العدد **4321**.

الوحدة الخامسة المصفوفات والمؤشرات **Arrays & Pointers**	.05



بنهاية هذه الوحدة:

- ستتعرف على بنية المصفوفات **(Arrays)** .
- ستتمكن من الإعلان عن وتمهيد والوصول إلى أي عنصر من عناصر المصفوفة .
- ستتعرف على المصفوفات متعددة الأبعاد.
- ستتمكن من استعمال المؤشرات **(Pointers)** .
- ستتمكن من استعمال مصفوفات السلاسل.

5.1 مقدمة

المصفوفة هي نوع من أنواع بنية البيانات ، لها عدد محدود ومرتب من العناصر التي تكون جميعها من نفس النوع type، فمثلاً يمكن أن تكون جميعها صحيحة int أو عائمة float ولكن لا يمكن الجمع بين نوعين مختلفين في نفس المصفوفة .

الشكل التالي يبين مصفوفة C تحتوى على 13 عنصر من النوع int، ويمكن الوصول إلي أي من هذه العناصر بذكر اسم المصفوفة متبوعاً برقم موقع العنصر في المصفوفة محاطاً بالأقواس [ ] .

يرمز لرقم العنصر في المصفوفة بفهرس العنصر index . فهرس العنصر الأول في المصفوفة هو 0 ولهذا يشار إلي العنصر الأول في المصفوفة C بـ C[0] والثاني C[1] والسابع C[6] وعموماً يحمل العنصر i في المصفوفة C الفهرس C[i-1] .

تتبع تسمية المصفوفات نفس قواعد تسمية المتغيرات.

C[0]	-45
C[1]	6
C[2]	0
C[3]	72
C[4]	1543
C[5]	-89
C[6]	0
C[7]	62
C[8]	-3
C[9]	1
C[10]	6453
C[11]	78
C[12]	15

أحياناً يسمى فهرس العنصر برمز منخفض subcript ويجب أن يكون الفهرس integer أو تعبير جبري تكون نتيجته integer . فمثلاً إذا كانت a=5 وb=6 فالعبارة:

```
C[a+b]+=2,
```

- تقوم بإضافة 2 إلي العنصر الثاني عشر C[11] في المصفوفة C .

- يحمل العنصر 0 في المصفوفة C القيمة 45- والعنصر 1 القيمة 6.

لطباعة مجموع الثلاثة عناصر الأولى في المصفوفة C يمكن كتابة:

```
cout<<C[0]+C[1]+C[2]<<endl;
```

5.2 المصفوفات Arrays

الإعلان عن المصفوفات:–

تحتل المصفوفات حيزاً في الذاكرة لذا يجب على المبرمج تحديد نوع عناصر المصفوفة وعددها حتى يتسنى للمعرف تخصيص الحيز اللازم من الذاكرة لحفظ المصفوفة ، وحتى تخبر المصرف بأن يخصص حيزاً لـ 12 عنصر من النوع int في مصفوفة C ، استخدم الإعلان:

```
int C[12];
```

يمكن تخصيص الذاكرة لعدة مصفوفات باستخدام نفس الإعلان وذلك كالأتي:

```
int b[100], x[20];
```

أيضاً يمكن الإعلان عن مصفوفات من أي نوع بيانات آخر ، فمثلاً للإعلان عن مصفوفة عناصرها من النوع char نكتب:

```
char ch[20];
```

مثال عن استخدام المصفوفات:

يستخدم البرنامج التالي حلقة for لتمهيد عناصر المصفوفة n عند 0 وطباعة عناصر المصفوفة.

```
//Program 5-1:
 //initializing an array
#include <iostream.h>
#include <iomanip.h>
main( )
{
int n[10];
  for (int i=0; i<10;i++) // initialize array
n[i] = 0;
cout << "Element" << setw(13) << " value" << endl;
for (i=0 ; i< 10; i++)   // print array
cout << setw(7) <<i<<setw(13) <<n[i]<<endl;
return 0;
}
```

الخرج من البرنامج:

```
Element              Value
                        0
                        0
                        0
                        0
                        0
                        0
                        0
                        0
                        0
     9                 0
```

في البرنامج السابق تم تضمين الملف **iomanip.h** وذلك لأننا استخدمنا المناور **setw(13)** والذي يعني ضبط عرض الحقل عند **13** (أي أن القيمة التي ستتم طباعتها ستكون على بعد **13** مسافة من القيمة التي تمت طباعتها قبلها) .

يمكن تمهيد عناصر المصفوفة باتباع الإعلان عن المصفوفة بعلامة المساواة (=) تليها لائحة من القيم المطلوب تمهيد عناصر المصفوفة عندها ، ويتم الفصل بين القيم بفواصل ، وتحيط هذه اللائحة الأقواس الحاصرة { }. البرنامج التالي يقوم بتمهيد عناصر من النوع **integer** لتحتوي قيم محددة عند الإعلان عن المصفوفة، وطباعة هذه القيم.

```
//Program 5-2:
 //initializing an array with a declaration
#include <iostream.h>
#include <iomanip.h>
main( )
{
int n[10] = {32,27,64,18,95,14,90,70,60,37};
cout << "Element" << setw(13) << " value" << endl;
for (i=0 ; i< 10; i++)   // print array
cout << setw(7) <<i<<setw(13) <<n[i]<<endl;
```

```
return 0;
}
```

ماذا يحدث إذا تم تحديد حجم مصفوفة لا يتوافق مع عدد قيم التمهيد الموجودة في اللائحة؟

إذا كانت قيم التمهيد الموجودة في اللائحة أكثر من حجم المصفوفة المحدد سيعترض المصرف، وإذا كانت أقل سيملأ المصرف بقية العناصر أصفار، لذا إذا كنا نريد تمهيد عناصر مصفوفة مهما كان حجمها بأصفار كل ما علينا فعله هو كتابة إعلان كالآتي:-

```
int anyarray[10]={0};
```

سيتم تمهيد العنصر الأول عند القيمة 0 التالي كتبناها والعناصر المتبقية عند 0 كوننا لم نحدد قيمة لها.

البرنامج التالي يقوم بجمع 12 عنصر في مصفوفة من النوع **int** .

```
//Program 5-3:
// compute the sum of the elements of the array
#include <iostream.h>
main( )
{
const int arraysize =12;
int a[arraysize] = {1, 3, 5, 4, 7, 2, 99, 16, 45, 67, 89, 45};
int total = 0;
for (int i= 0; i<arraysize ; i++)
total += a[i];
cout <<" total of array element values is " << total << endl;
return 0;
}
```

الخرج من البرنامج:

```
total of array element values is 383
```

نلاحظ أننا في العبارة:

const int arraysize = 12;

استعملنا كلمة جديدة هي **const** . يتم استعمال هذه الكلمة الأساسية في تعريف المتغير الذي لا يمكن تغيير قيمته في البرنامج ولذلك يجب تمهيده عند قيمة أولية عند تعريفه (في البرنامج السابق تم تمهيده ليساوى 12)

.35 السلاسل Strings

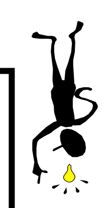
كما ذكرنا أنه يمكن تعريف مصفوفات من أي نوع بيانات آخر، سنقوم الآن بتخزين سلسلة حروف في مصفوفة من النوع char.

يتم تمهيد المصفوفة من النوع char باستخدام ما يسمى بالثابت السلسلي (string literal)

```
char string1[ ]="first";
```

حجم المصفوفة string1 هنا يتم تحديده بواسطة المصرف بناءاً على طول الثابت السلسلي "first".

من المهم هنا أن نذكر أن السلسلة "first" تحتوى على خمسة عناصر زائداً حرفاً خامداً يشير إلى نهاية السلسلة ويسمى الحرف الخامد null character ويتم تمثيله باستخدام تتابع الهروب '\o' وتنتهي كل السلاسل بهذا الحرف الخامد وعليه فإن المصفوفة string1 تحتوى على ستة عناصر.



يجب أن نتذكر دائماً أن المصفوفة التالية تعلن عنها ثوابت سلسلية يجب أن تكون كبيرة لما يكفى لتخزين حروف السلسلة إضافة إلى الحرف الخامد.

يمكن أيضاً تمهيد السلسلة "first" باستخدام لائحة قيم تفصلها فواصل لذا الإعلان:-

```
char string1[ ]="first";
```

يكافئ:

```
char string1[ ]={'f','i','r','s','t','\o'}
```

وبما أن السلسلة في الواقع هي مصفوفة أحرف ، عليه يمكن الوصول إلى أي حرف من حروف السلسلة مباشرة باستخدام الفهرس واسم المصفوفة ،فمثلاً string1[0]='f'.

ومثلما يمكن تمهيد السلسلة عند الإعلان عنها ، يمكن أيضاً إدخال السلاسل عن طريق لوحة المفاتيح باستعمال cin و>> فمثلاً الإعلان :-

```
char string2[20];
```

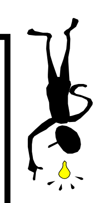
ينشئ مصفوفة أحرف تسمح بتخزين 19 حرفاً إضافة إلى الحرف الخامد والعبارة

```
cin>>string2;
```

تقوم بتخزين السلسلة المدخلة عن طريق لوحة المفاتيح وتخزينها في المصفوفة string2.

يمكن خرج السلسلة المخزنة في مصفوفة الأحرف باستخدام cout و<< وعليه يمكن طباعة المصفوفة string2 باستخدام العبارة:-

```
cout << string2 << endl;
```



عند استعمال cin مع السلاسل يتم فقط ذكر اسم المصفوفة التي سيتم فيها تخزين حروف السلسلة المدخلة دون ذكر حجمها هنا تأتى مسئولية المبرمج في أمثلة المصفوفة التي سيتم تعريفها لتخزين السلسلة يجب أن تكون كبيرة لما يكفى تخزين السلسلة التي يدخلها المستخدم عن طريق لوحة المفاتيح ويجب أن نذكر هنا أن cin حالما يجد فراغاً يتوقف عن قراءة الدخل ويقوم بتخزين السلسلة المدخلة في المصفوفة المعلن عنها لتخزينها.

cout مثل cin لا تهتم بحجم المصفوفة حيث تقوم بطباعة حروف السلسلة حتى تصل إلى الحرف الخامد الذي يحدد نهاية السلسلة.

البرنامج التالي يقوم بتمهيد مصفوفة أحرف عند ثابت سلسلي ويقوم باستعمال حلقة التكرار for للوصول إلى عناصر المصفوفة وطباعتها .

```
//Program 5-4:
//Treating character arrays as strings
#include<iostream.h>
main( )
{
char string1[20], string2[ ] = " stringliteral" ;
cout << "Enter a string: ";
cin>> string1;
cout << "string1 is : " << string1<<endl
      << "string2 is : " << string2<<endl
      << "string1 with spaces between characters is: "
```

```
        << endl;
for (int i= 0; string1[i] ; = '\0' ; i++)
   cout << string1[i]<< '  ';
cout << endl;
//Continued
return 0;
}
```

الخرج من البرنامج:

بافتراض أن المستخدم قد أدخل السلسلة Hello there



```
Enter a string: Hello there
string1 is : Hello
string2 is : string Literal
string1 with spaces between characters is : H  e  l  l  o
```

استخدمت حلقة التكرار for لوصول إلى حروف السلسلة string1 وطباعتها مع طباعة مسافة بين كل حرف والآخر حتى تصل إلى الحرف الخامد '\o' (string1[i] != '\o';) والذي يحدد نهاية السلسلة.

5.4 مكتبة دالات السلاسل

توجد عدة دالات تعمل على السلاسل، إذا أردنا استعمال أي من هذه الدوال في برنامج يجب أن نقوم بتضمين ملف الترويسة string.h . من هذه الدالات :

1/ strlen( ):-

تعيد الدالة strlen( ) طول السلسلة الممررة كوسيطة لها ،البرنامج التالي يوضح ذلك :-

```
//Program 5-5:
// using strlen
#include<iostream.h>
#include<string.h>
```

```
main ( )
{
char *string1= " abcdefghijklmnopqrstuvwxyz";
//Continued
char *string2 = "four";
char *string3 = "Boston";
cout << " The length of \ "  " << string1
       << " \" is  << strlen (string1) <<endl
       << " The length of \" << string2
       <<" \" is << strlen (string2) << endl
       << "The length of\ "  "<< string3
      << " \" is << strlen( string3) <<endl;
return 0;
}
```

الخرج من البرنامج:

```
The length of "abcdefghijklmnopqrstuvwxyz" is 26
The length of "four" is 4
The length of "Boston" is 6
```

لاحظ أن الحرف **0** غير محسوب في الطول الذي تعيده الدالة **strlen** على الرغم من أنه موجود في **s1** ويحتل مكاناً في الذاكرة.

2/ **strcpy( )**:-

تستعمل الدالة **strcpy** لنسخ سلسلة إلى سلسلة أخرى

```
//Program 5-6:
// using strcpy
#include<iostream.h>
```

```
#include<string.h>
main ( )
{
char x[ ] = "Happy Birthday to you";
//Continued
char y[25];
cout<<" The string in array x is : "<< x << endl;
cout<<" The string in array y is : "<< strcpy(y, x)
<< endl;
return 0;
}
```

بعد تنفيذ العبارة strcpy(y, x) ستحتوى السلسلة y على Happy Birthday to you. لاحظ هنا أن الدالة strcpy تنسخ السلسلة الممررة كالوسيطة الثانية إلى السلسلة الممررة كالوسيطة الأولى.

وعليه الخرج من البرنامج:

```
The string in array x is : Happy Birthday to you
The string in array y is : Happy Birthday to you
```

3/ strcat( ):-

تقوم الدالة strcat( ) بإلحاق السلاسل ، الذي يمكن أن يسمى جمع السلاسل فمثلاً إذا ألحقنا السلسلة science بالسلسلة computer ستكون نتيجة السلسلة computer science:-

```
//Program 5-7:
// using strcat
#include<iostream.h>
#include<string.h>
int main ( )
```

```
{
char s1[20]="computer" ;
char s2[  ]="science" ;
cout<<"s1= " <<s1 << endl << "s2= " << s2 <<endl;
cout<< "strcat(s1, s2) = " << strcat (s1, s2) << endl;
//Continued
return 0;
}
```

الخرج من البرنامج:

```
s1= computer
s2 = science
strcat(s1, s2)= computerscience
```

4/ **strcmp( )**:-

الدالة **strcmp** تقارن السلسلة الممرة إليها كوسيطة أولى مع السلسلة الممرة إليها كوسيطة ثانية، وترجع 0 إذا كانتا متطابقتين وقيمة سالبة إذا كانت السلسلة الأولى أصغر من السلسلة الثانية وقيمة موجبة إذا كانت السلسلة الأولى أكبر من السلسلة الثانية.

البرنامج التالي يوضح ذلك:

```
//Program 5-8:
// using strcmp
#include<iostream.h>
#include<string.h>
int main (  )
{
char  *s1 = " Happy New Year";
char  *s2 = " Happy New Year";
char  *s3 = " Happy Holidays";
cout << "s1= " << s1<< endl<< "s2= " <<s2<<endl
```

```
    << "s3= " << s3<< endl<< endl<< "strcmp(s1, s2)= "
    << strcmp(s1, s2) <<endl<< "strcmp(s1, s3)= "
    << strcmp(s1, s3) <<endl<< "strcmp(s3, s1)= "
    << strcmp(s3, s1) <<endl<< endl;
return 0;
}
```

الخرج من البرنامج:

```
s1=   Happy New Year
s2=   Happy New Year
s3 =  Happy Holidays

strcmp (s1, s2) = 0
strcmp (s1, s3) = 6
strcmp (s3, s1) = 6
```

5.5 تمرير المصفوفات كوسائط للدوال Passing Arrays to Functions

يمكن تمرير مصفوفة كوسيطة لدالة وذلك بذكر اسم المصفوفة.

مثلاً إذا تم الإعلان عن مصفوفة **hourlyTemperature** كالآتي:-

```
int hourlyTemperatures[24];
```

عبارة استدعاء الدالة:-

```
modify_Array(Int hourlyTemperatures,24);
```

تمرر المصفوفة hourlyTemperature وحجمها كوسائط للدالة modify Array وتذكر دائماً أنه عند تمرير مصفوفة ما كوسيطة لدالة يجب تمرير حجم المصفوفة حتى يتسنى للدالة معالجة كل عناصر المصفوفة.

5.6 المصفوفات متعددة الأبعاد Multidimensional Arrays

يمكن للمصفوفات في C+ + أن تكون متعددة الأبعاد ويمكن كذلك أن يكون كل بعد بحجم مختلف ، الاستعمال الشائع للمصفوفات متعددة الأبعاد هو تمثيل الجداول Tables التالي تحتوي على بيانات مرتبة في صورة صفوف وأعمدة ولتمثيل الجدول نحتاج لبعدين الأول يمثل الصفوف والثاني يمثل الأعمدة.

الشكل التالي يبين مصفوفة A تحتوى على ثلاثة صفوف وأربع أعمدة.

	Column 0	Column1	Column2	Column 3
Row 0	A[0][0]	A[0][1]	A[0][2]	A[0][3]
Row 1	A[1][0]	A[1][1]	A[1][2]	A[1][3]
Row 2	A[2][0]	A[2][1]	A[2][2]	A[2][3]

يتم تمثيل أي عنصر في المصفوفة A على الصورة A[i][j] حيث:-

A : اسم المصفوفة.

i : رقم الصف الذي ينتمي إليه العنصر.

j : رقم العمود الذي ينتمي إليه العنصر.

لاحظ أن كل العناصر الموجودة في الصف الأول مثلاً يكون الفهرس الأول لها هو 0 وكل العناصر الموجودة في العمود الرابع يكون الفهرس الثاني لها هو 3.

يتم الإعلان عن مصفوفة a تحتوى على x صف و y عمود هكذا:

```
int a[x][y];
```

يمكن تمهيد قيمة المصفوفة المتعددة الأبعاد عند الإعلان عنها وذلك كالآتي:

```
int b[2][2]={{1,2},{3,4}};
```

حيث:

b[1][1]=4,   b[1][0]=3,   b[0][1]=2,   b[0][0]=1

أيضاً هنا في المصفوفة متعددة الأبعاد إذا تم تمهيدها عند قيم لا يتوافق عددها مع حجم المصفوفة فإن المصرف سيملأ بقية العناصر أصفار.

البرنامج التالي يوضح كيفية تمهيد مصفوفات متعددة الأبعاد عند الإعلان عنها:

```
//Program 5-9:
// initializing multidimensional arrays
#include<iostream.h>
void printarray(int [ ] [3]);
```

```
int main( )
//continued
{
int  array1[2] [3] = { {1, 2, 3}, {4, 5, 6}},
      array2[2] [3] = {1, 2, 3, 4, 5},
      array3[2] [3] = { {1, 2}, {4} };
cout << "values in array1 by row are : " << endl;
printArray(array1);
//Continued
cout << "values in array2 by row are : " << endl;
printArray(array2);
cout << "values in array3 by row are : " << endl;
printArray(array3);
return 0;
}
void printArray(int a[ ][3])
{
for (int i=0; i<1; i++) {
 for (int j=0; j<2; j++)
cout << a[i][j] <<' ';
cout << endl;
         }
   }
```

الخرج من البرنامج:

```
values in array 1 by row are:
1   2   3
4   5   6
values in array 2 by row are:
1   2   3
4   5   0
values in array 3 by row are:
1   2   0
```

4 0 0

5.7 المؤشرات Pointers

يستخدم المؤشر في لغة + +C كعنوان لمتغير في الذاكرة ، أحد الاستعمالات المهمة للمؤشرات هو التخصيص الديناميكي للذاكرة حيث يتم استعمال المؤشرات لإنشاء بنية بيانات لتخزين البيانات في الذاكرة.يتم الإعلان عن المؤشرات قبل استخدامها في البرنامج فمثلاً العبارة :

```
int *countptr;
```

تعلن عن مؤشر countptr ليشير إلى متغير من النوع int (* المذكورة قبل اسم المؤشر تشير لذلك) وكل متغير يعلن عنه كمؤشر يجب أن يكتب في الإعلان مسبوقاً بـ * فمثلاً الإعلان :

```
float *xptr, *yptr;
```

يشير لأن كلاً من xptr و yptr موقعي مؤشرات لقيم من النوع float ويمكن أن تستخدم المؤشرات لتشير لأي نوع بيانات آخر.

تذكر دائماً عند الإعلان عن أي مؤشر أن تسبق *كل مؤشر على حدة فمثلاً الإعلان :

Int *xptr, yptr; ليس صحيحاً.

يجب أن تعلن عن هذه المؤشرات كالآتي:

```
int *xptr, *yptr;
```

يمكن تمهيد المؤشرات عند الإعلان عنها عند قيمة 0 أو null أو عند قيمة عنوان في الذاكرة . المؤشر الذي يحمل القيمة 0 أو null لا يشير لأي متغير . تمهيد المؤشر عند 0 يكافئ تمهيده عند null ولكن في + +C يفضل تمهيد المؤشر عند القيمة 0.

عوامل المؤشرات:-

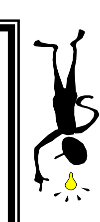
1/ عامل العنوان &:-

العامل & يسمى عامل العنوان وهو عامل أحادى يستعمل لمعرفة العنوان الذي يحتله متغير ما [يرجع عنوان معامله] فمثلاً إذا استعملنا الإعلان:

```
int y= 5;
int *yptr;
```

العبارة: yptr =&y;

تقوم بتعيين عنوان المتغير y للمؤشر yptr ويقال أن yptr يشير لـ y .



إنتبه للفرق بين عامل العنوان & الذي يسبق اسم المتغير، وبين عامل المرجع الذي يلي اسم النوع في تعريف الدالة.

2/ العامل * :

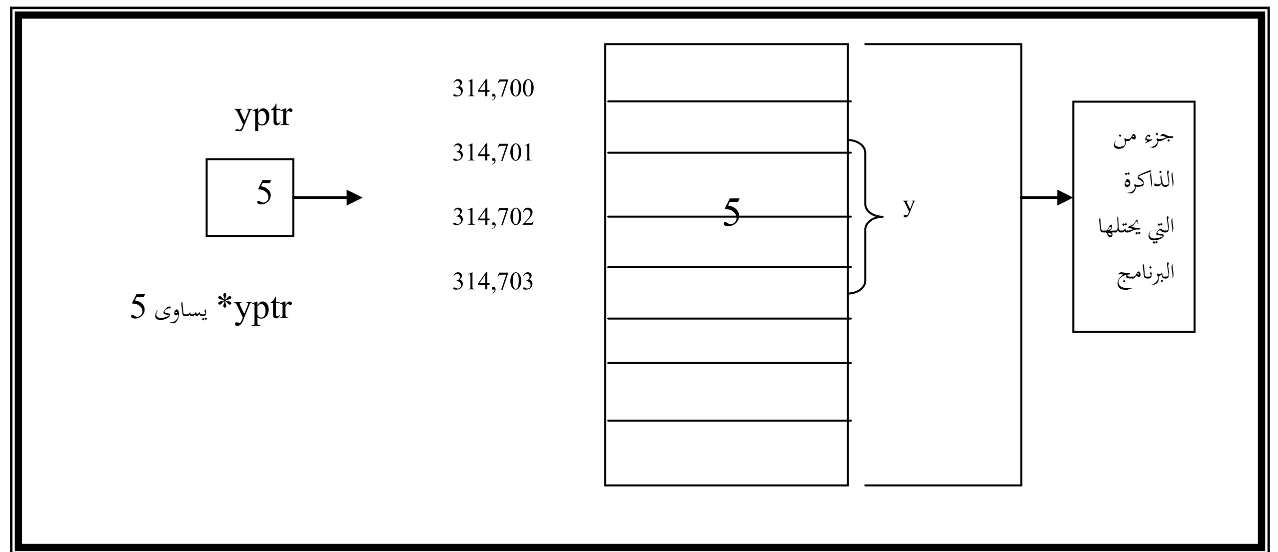
العامل * أيضاً عامل أحادى وهو يرجع القيمة التي يحملها معامله ، وعليه العبارة

```
cout << * yptr << endl ;
```

تقوم بطباعة قيمة المتغير y والتي هي 5 .

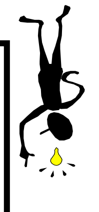
والعبارة: cout<<yptr; تقوم بطباعة القيمة 314,701 والتي هي عنوان المتغير y ، بعد أن تم تعيين المتغير y إلى yptr .

الشكل(5-1) يبين هذا:



شكل (5-1) يوضح المخرج من yptr*

وعندما يتم استعمال العامل * على يسار اسم المتغير كما حصل في التعبير yptr* فإنه يسمى عامل المواربة indirection.



العامل * عند استعماله كعامل مواربة له معنى مختلف عن معناه عند استعماله للإعلان عن المتغيرات المؤشرة. يسبق عامل المواربة اسم المتغير ويعنى قيمة المتغير المشار إليه. أما * المستعملة في الإعلان فتعنى مؤشر إلى.

```
Int *yptr ; ( إعلان)
```

*yptr=5; (موازية)

البرنامج يوضح استعمال العامل & والعامل * .

```
//Program 5-10:
// using the & and * operators
#include<iostream.h>
main ( )
{
  int a ;                          //a is an integer
  int  *aptr;                     // aptr is apointer to an integer
   a = 7;
   aptr = &a;                // aptr set to address of a
   cout <<" The address of a is " << &a <<endl
          << "The value of aptr is " << aptr<< endl<< endl;

  cout << "The value of a is " << a<< endl
       << "The value of *aptr is " << *aptr<< endl<<endl;
  cout<<" Proving that * and & are complement of "
        << "each other." <<endl<< " & *ptr = "<< & *aptr
        << endl<< " *&aptr = " << *&aptr <<endl;
   return  0;
}
```

الخرج من البرنامج:

```
The address of a is oxfff4
The value  of aptr is oxfff4

The value of a is 7
The value of *aptr is 7

Proving that * and & are complements of each other
&* aptr = oxfff4
*& aptr = oxfff4
```

مؤشرات إلى void:-

عادة العنوان الذي نضعه في المؤشر يجب أن يكون من نفس نوع المؤشر، فمثلاً لا يمكننا تعيين عنوان متغير **float** إلى مؤشر **int** ، لكن هنالك نوع من المؤشرات يمكنها أن تشير إلى أي نوع من البيانات وتسمى مؤشرات إلى void ويتم تعريفها كالآتي:-

void * ptr;

لهذا النوع من المؤشرات استعمالات خاصة فهو يستخدم مثلاً لتمرير المؤشرات إلى دالات تعمل على عدة أنواع بيانات.

المثال التالي يبين أنه إذا لم يتم استعمال مؤشرات إلى void يجب أن نعين للمؤشر عنواناً من نفس نوعها:

```
//Program 5-11:
#include<iostream.h>
void main( )
int  intvar;
float flovar;
int* ptrint;
void* ptrvoid;
ptr* ptrflovar;
ptrint=&intvar;
// ptr int = &flovar; //Error
// ptr flo = &intvar; //Error
```

```
ptrvoid=&intvar;
ptrvoid=&flovar;
}
```

في المثال السابق يمكن تعيين عنوان المتغير intvar إلي المؤشر ptr int لأنهما من النوع *int لكن لا يمكننا تعيين عنوان المتغير flovar إلى المؤشر ptrint لأن الأول من النوع *float والثانى من النوع *int . لكن يمكن تعيين أي نوع مؤشرات إلى المؤشر ptrvoid لأنه مؤشر إلى void.

5.8 المؤشرات في استدعاء الدوال

هنالك ثلاث طرق لتمرير الوسائط للدوال :-

١ -التمرير بالقيمة call-by-value .

٢ - التمرير بالمرجع call-by-reference .

٣ - التمرير بالمرجع مع مؤشر call by reference with pointer arguments.

كما ذكرنا سابقاً أن العبارة **return** تستعمل لإعادة قيمة من دالة مستدعاة ورأينا أيضاً أنه يمكن تمرير الوسائط للدوال بالمرجع حتى يتسنى للدالة التعديل في البيانات الأصلية للوسائط ،يستخدم مبرمجو ++C المؤشرات لمحاكاة استدعاء الدوال بالمرجع . عند استدعاء الدالة يتم تمرير عنوان الوسيطة ويتم ذلك بكتابة عامل العنوان للوسيطة المطلوب معالجتها . عندما يتم تمرير عنوان الوسيطة للدالة يتم استعمال العامل * للوصول لقيمة المتغير .

البرنامجان أدناه يحتويان على إصدارين من دالة تقوم بتكعيب عدد صحيح.

```
//Program 5-12:
// Cube a variable using call-by-value
#include<iostream.h>
int cubeByValue(int);        // prototype
int main( )
{
int number = 5;
cout <<" The original value of number is "
     <<number<<endl;
number = cubeByValue(number);
cout << " The new value of number is " << number<< endl;
return 0;
}
int cubeByValue(int n)
```

```
{
return n*n*n; // cube local variable n
}
```

الخرج من البرنامج:

```
The original value of number is 5
The new value of number is 125
```

يقوم هذا البرنامج بتمرير المتغير كوسيطة للدالة مستخدماً طريقة التمرير بالقيمة حيث تقوم الدالة **cubebyvalue** بتكعيب المتغير **number** وتقوم بإرجاع النتيجة للدالة **main** باستخدام العبارة **return** .

في البرنامج التالي يتم تمرير عنوان المتغير **number** كوسيطة للدالة **cube by reference** حيث تقوم الدالة بتكعيب القيمة التي يشير إلي المؤشر **nptr**.

```
//Program 5-13:
// cube a variable using call-by-reference with a
pointer argument
#include<iostream.h>
void cubeByReference (int *); // prototype
main( )
{
int number = 5;
cout<< " The original value of number is " << number
        <<endl;
cubeByReference(&number);
cout<< " The new value of number is " << number <<endl;
return 0;
}
void cubeByReference (int *nPtr)
{
*nPtr = *nPtr * *nPtr * *nPtr;      // cube number in
main
```

```
}
```

الخرج من البرنامج:

```
The original value of number is 5
The new value of number is 125
```

نذكر هنا أن الدالة التي يتم تمرير عنوان متغير كوسيطة لها يجب أن يتم فيها تعريف مؤشر يحمل قيمة العنوان ، فمثلاً في الدالة **cubeByReference**:-

```
void cubeByReference (int *nptr)
```

المصرح في الدالة **cubeByReference** يشير إلى أنه سيتم تمرير عنوان لمتغير من النوع **integer** كوسيطة لها ويتم تخزين العنوان في المؤشر **nptr** وهى لا ترجع قيمة للدالة **main**.

وكما ذكرنا سابقاً أنه في الإعلان عن الدالة يكفى فقط ذكر نوع المتغير الذي سيتم تمريره كوسيطة للدالة دون ذكر اسم المتغير ثم الإعلان عن الدالة **cube by reference** كالآتي:-

```
void cubeByReference (int *)
```

5.9 المؤشرات والمصفوفات

عرفنا سابقاً كيف يمكن الوصول إلى العناصر المخزنة في المصفوفات باستعمال اسم المصفوفة وفهرس العنصر. المثال التالي يوضح هذا:

```
int array1[3]={1,2,3};
for (int j=0;j<3;j++)
cout<<endl<<array1[j];
```

يعرض الجزء السابق عناصر المصفوفة array1 كالآتي:

```
1
2
3
```

يمكن الوصول إلي عناصر المصفوفات أيضاً باستخدام المؤشرات.

المثال التالي يوضح كيف يمكن الوصول إلى عناصر نفس المصفوفة السابقة باستعمال المؤشرات:

```
int array1[3]={1,2,3};
for (int j=0;j<3;j++)
cout<<endl<< *(array1+j);
```

أيضاً يعرض هذا الجزء:

```
1
2
3
```

التعبير ;*(array1+j) له نفس تأثير التعبير array1[j] وذلك للآتي:

افرض أن j=1 لذا يكون التعبير *(array1+j) مرادفاً للتعبير *(array1+1) ويمثل هذا محتويات العنصر الثاني في المصفوفة array1 وإن اسم المصفوفة يمثل عنوانهاوهو عنوان أول عنصر في المصفوفة، ولهذا فالتعبير array+1 يعنى عنوان العنصر الثاني في المصفوفة و array1+2 يعنى عنوان العنصر الثالث في المصفوفة ، ولكننا نريد طباعة قيم عناصر المصفوفة array وليس عناوينها، لهذا استعملنا عامل المواربة للوصول إلى قيم عناصر المصفوفة.

5.10 المؤشرات والسلاسل

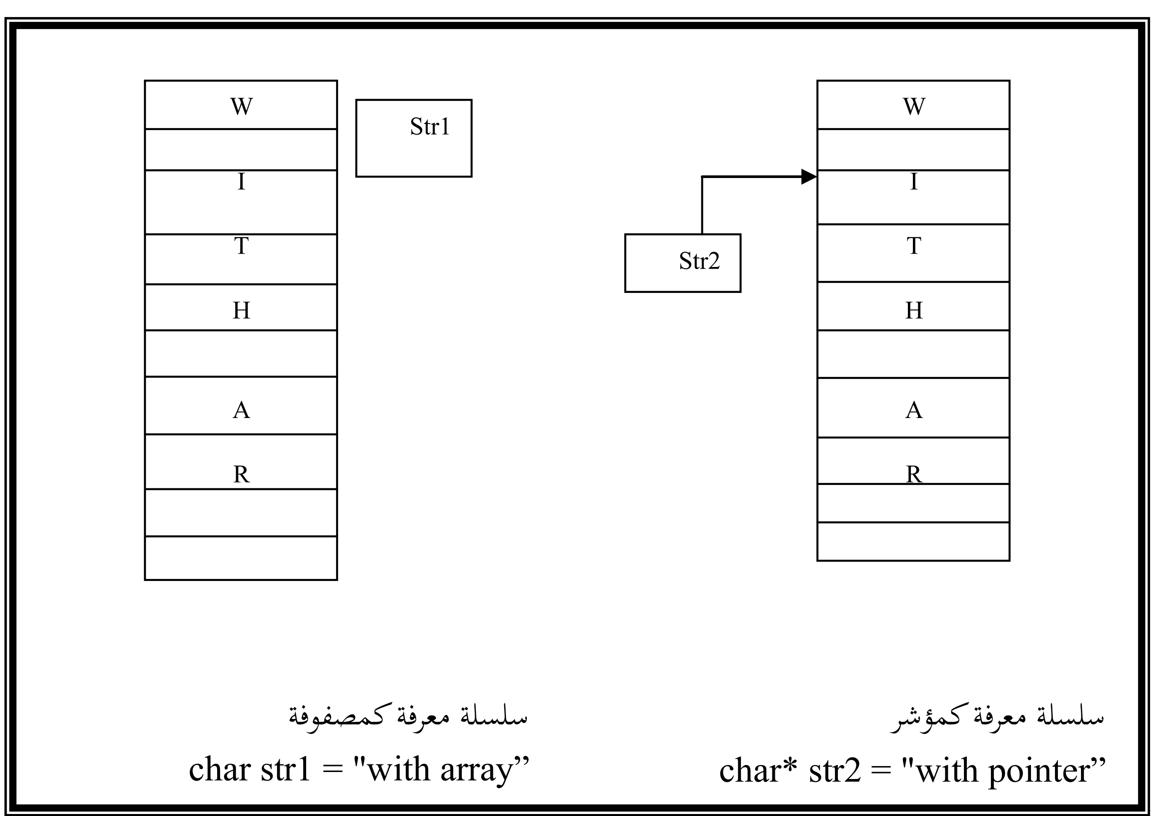
كما ذكرنا سابقاً السلاسل هي مجرد مصفوفات من النوع **char** لذا يمكننا استخدام المؤشرات مع أحرف السلاسل مثلما يمكن استخدامه على عناصر أي مصفوفة.

المثال التالي يتم فيه تعريف سلسلتين واحدة باستعمال المصفوفات كما في أمثلة السلاسل السابقة والأخرى باستعمال المؤشرات:

```
char str1[ ] ="with array";
char str2[ ] ="with pointer";
cout <<endl<<str1;
cout <<endl<<str2;
str2++;
cout <<endl<<str2;
```

تتشابه السلسلتان السابقتان في عدة نواحي إلا أن هنالك فرق مهم : **str1** هو عنوان أي ثابت مؤشر بينما **str2** هو متغير مؤشر.

الشكل (5-2) يبين كيف يبدو هذان النوعان في الذاكرة:



شكل (5-2) يوضح محتوى الذاكرة في حالتي التعريف كسلسلة والتعريف كمؤشر

لذا يمكننا زيادة str2 لأنه مؤشر ولكن بزيادته سيشير إلى الحرف الثاني في السلسلة وعليه الخرج من المثال السابق:-

```
with array
with pointer
ith pointer
```

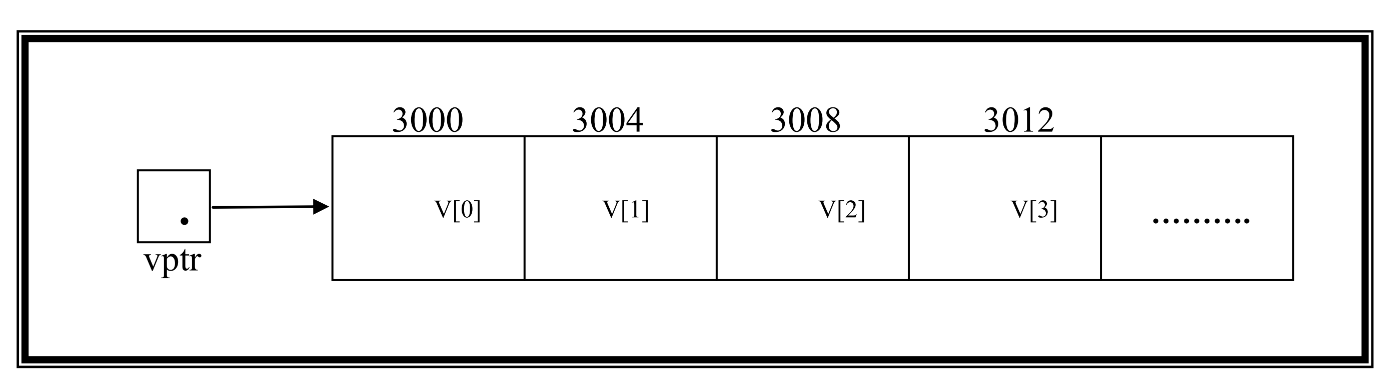
5.11 استعمال العوامل الحسابية مع المؤشرات

يمكن أن تكون المؤشرات معاملات في التعابير الحسابية وفى تعابير التعيين والتعابير العلائقية . سنتطرق هنا للعوامل التي يمكن أن تكون المؤشرات معاملات لها وكيفية استعمال هذه العوامل مع المؤشرات .

يمكن استعمال (++) أو (--) لزيادة أو نقصان المؤشرات بمقدار واحد كما يمكن أيضاً إضافة متغير صحيح للمؤشر عن طريق استعمال العامل (+) أو العامل (=+) ويمكن نقصان متغير صحيح من مؤشر عن طريق استعمال (-) أو (=-) كما يمكن أيضاً نقصان أو زيادة مؤشر لمؤشر آخر.

افترض أنه تم الإعلان عن مصفوفة int v[10] ، يحمل العنصر الأول في المصفوفة العنوان 3000 في الذاكرة.

افترض أيضاً أنه تم تمهيد مؤشر vptr ليشير للعنصر الأول في المصفوفة v[0] وعليه قيمة المؤشر vptr هي 3000 ، الشكل (3-5) يبين هذا:-



شكل (3-5) تمهيد vptr ليشير للعنصر الأول في المصفوفة

يمكن تمهيد المؤشر vptr ليشير للمصفوفة v بإحدى العبارتين التاليتين:

```
vptr = v;
vptr = & v[0];
```

عنوان العنصر v[0] في المصفوفة v هو 3000 وعنوان العنصر v[1] هو 3004 وذلك لأن عناصر المصفوفة v هي عبارة عن متغيرات صحيحة integer واستخدام تمثل 4bytes من الذاكرة، وعليه عند إضافة أو طرح متغير صحيح integer من مؤشر تتم إضافة المتغير مضروباً في حجم المتغير في الذاكرة والثاني يعتمد على نوع المتغير حيث يحتل المتغير الصحيح كما ذكرنا 4bytes والمتغير الحرفي char يحتل 1byte وعموماً يعتمد ذلك على عدد الـbytes التالي يحتلها المتغير ، فمثلاً العبارة التالية :

```
vptr +=2;
```

تؤدى لإضافة 8 للمؤشر vptr بافتراض أن المتغير الصحيح يحتل 4bytes من الذاكرة.

إدارة الذاكرة باستعمال العوامل **new** و **delete**:-

تستعمل المصفوفة لتخزين عدد من الكائنات أو المتغيرات فالعبارة:

```
int ar1[50];
```

تحجز الذاكرة ل 50 عدد صحيح فالمصفوفات هي أسلوب مفيد لتخزين البيانات لكن لها عائق مهم : علينا معرفة حجم المصفوفة في وقت كتابة البرنامج . في معظم الحالات قد لا نعرف كمية الذاكرة التالي سنحتاج إلي أثناء تشغيل البرنامج.

تزود *C++* أسلوباً خاصاً للحصول على كتل من الذاكرة :

العامل **new**:-

يخصص العامل new كتل ذاكرة ذات حجم معين ويعيد مؤشراً لنقطة بداية كتلة الذاكرة تلك، يحصل العامل new على الذاكرة ديناميكياً أثناء تشغيل البرنامج .

الصورة العامة لكتابة العامل new هي:

```
p-var = new type;
```

حيث:-

p-var: متغير مؤشر يتم فيه تخزين عنوان بداية كتلة الذاكرة المخصصة بواسطة العامل new تسمح بتخزين متغير من النوع type .

العامل **delete**:-

إذا تم حجز العديد من كتل الذاكرة بواسطة العامل **new** سيتم في النهاية حجز كل الذاكرة المتوفرة وسيتوقف الحاسوب عن العمل . لضمان استعمال آمن وفعال للذاكرة يرافق العامل **new** عامل يسمى **delete** يعيد تحرير الذاكرة لنظام التشغيل .

الجزء من البرنامج التالي يبين كيف يتم الحصول على ذاكرة لسلسلة :

```
char * str=" It is the best.";
int len = strlen(str);
char*ptr;
ptr= new char[len+1];
strcpy(ptr,str);
cout<<"ptr="<<ptr;
delete[ ] ptr ;
```

تم استعمال الكلمة الأساسية **new** يليها نوع المتغيرات التي سيتم تخصيصها وعدد تلك المتغيرات ، يقوم المثال بتخصيص متغيرات من النوع **char** ويحتاج إلى **len+1** منها حيث تساوي **len** طول السلسلة **str** ، الرقم **1** ينشئ بايتاً إضافياً للحرف الخامد الذي ينهي السلسلة ويعيد العامل **new** مؤشراً يشير إلى بداية قطعة الذاكرة التي تم تخصيصها.تم استعمال المعقفات للدلالة على أننا نخصص ذاكرة لمصفوفة .

```
ptr =new char[len+1];
```

العبارة:

```
delete [ ] ptr;
```

تعيد للنظام كمية الذاكرة التي يشير إليها المؤشر **ptr**.

المعقفات [ ] التي تلي العامل **delete** تشير لأننا نقوم بحذف مصفوفة، لا نحتاج لاستعمالها إذا كنا نقوم بحذف متغير واحد بواسطة العامل **delete**.

المؤشر **This**:

يمتلك كل كائن في فئة مؤشراً خاصاً يسمى **this** يشير إليه، وباستخدام هذا المؤشر يستطيع أي عضو دالي في الفئة معرفة عنوان الكائن الذي استدعاه .

المثال التالي يوضح هذا :-

```
//Program 5-14:
#include<iostream.h>
```

```
class where
 {
  private:
       char  chararray[10];
  public:
//Continued
       void reveal( )
{ cout <<"My Objects address is "<<this;
};

main( )
{
 where w1,w2;
 w1.reveal( );
 w2.reveal( );
}
```

ينشئ هذا البرنامج كائنات من النوع **where**، ويطلب من كل منها عرض عنوانه باستعمال الدالة **reveal( )**، والتي تعرض قيمة المؤشر **this**.

الخرج من البرنامج يبدو كالتالي:

```
My object's address is ox8f4effec
My object's address us ox8f4effe2
```

نلاحظ إن عنوان الكائن **w2** يبتعد **Bytes 10** عن عنوان **w1**، وذلك لأن البيانات في كل كائن تتألف من مصفوفة من **10 Bytes**.

يمكن معاملة المؤشر **this** كأي مؤشر كائنات آخر، لذا يمكن استخدامه للوصول إلى بيانات الكائن الذي يشير إليه كما هو مبين في البرنامج أدناه.

```
//Program 5-15:
#include<iostream.h>
class test {
 public:
```

```
  test(int=0);
  void print( ) const;
 private:
  int x;
 };
void test::print( ) const
//Continued
{
  cout <<" X="<<x<<endl
<<"this-> x= "<<this->x<<endl;
     <<"(*this).x="<<(*this).x<<endl;
}
main ( )
{
 test a(12);
 a.print( );
 return 0;
}
```

وللتوضيح فإن العضو الدالي print يقوم أولاً بطباعة x مباشرة، ثم يستعمل طريقتين للوصول إلى x باستعمال المؤشر this:-

الأولى: باستعمال العامل (->).

الثانية: باستعمال العامل (.).

لاحظ الأقواس التي تحيط بـ this*، عندما نقوم باستخدام العامل (.) للوصول إلى أعضاء الفئة نستعمل الأقواس، وذلك لأن العامل (.) له أولوية أعلى من العامل *، وعليه بدون الأقواس يتم تقييم التعبير this.x* كالآتي:

*(this.x)

والذي ينتج عرض رسالة خطأ من المصرف لأن العامل (.) لا يستخدم مع المؤشرات.

هنالك استعمالات أخرى للمؤشر this سنتطرق لها عند تحميلنا للعوامل بشكل زائد.

الملخص:

- المصفوفة هي عبارة عن مجموعة متتابعة من العناصر المحدودة التي تكون جميعها من نفس نوع البيانات.
- يعلن عن المصفوفات تحديد نوع عناصر المصفوفة ثم اسم المصفوفة متبوعاً بعدد العناصر فيها بين قوسين [ ]، فمثلاً لتخزين مائة عنصر من النوع int في مصفوفة b نكتب :

```
int b[100];
```

- تستخدم المصفوفات من النوع char لتخزين سلاسل الأحرف.
- يمكن تمهيد مصفوفة أحرف عند ثابت سلسلي كالآتي:

```
char a[10] = "computer";
```

- تنتهي كل السلاسل بحرفاً خاصاً يسمى بالحرف الخامد والذي يتم تمثيله بتتابع الهروب ('\0').
- يمكن تمهيد السلاسل باستخدام لائحة قيم كالآتي:

```
char a[10] = {'c', 'o', 'm', 'p', 'u', 't', 'e', 'r' , '\0'};
```

- تعيد الدالة strlen( ) طول السلسلة الممرة كوسيطة لها.
- تستخدم الدالة strcpy( ) لنسخ سلسلة إلى سلسلة أخرى.
- تقوم الدالة strcat( ) بإلحاق السلاسل.
- تقارن الدالة strcmp( ) بين سلسلتين.
- المؤشرات هي عبارة عن متغيرات تستخدم كعناوين للمتغيرات في الذاكرة.



الأسئلة

1/ أكتب عبارات *C++* تقوم بالآتي:

١ – طباعة العنصر السابع في مصفوفة أحرف تدعى f.

٢ – إدخال قيمة العنصر الرابع في مصفوفة أعداد صحيحة b.

2/ ما هو الخطأ في العبارات التالية:

```
a\ char str [5];
       cin >>str;   // user types hello
b\ int a[3];
       cout <<a[1] << "  " << a[2]<<"   "<< a[3] <<endl;
c\ float  f[3] = { 1.1 , 10.01, 100,001, 1000.0001  } ;
d\ double d[2][10];
    d[1, 9] = 2.345;
```

3/ ما الغرض من البرنامج التالي:

```
#include <iostream.h>
  int WhatIsThis (int[ ] ,int);
main
{
const int  arraysize = 10;
int a[arraysize] = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };
int result = WhatIsThis (q, arraysize);
cout << " Result is: "  << result << endl;
return 0;
}
       int WhatIsThis (int b[ ], int size)
       {
              if (size == 1)
              return b[0];
              else
                  return b[size -1] +WhatIsThis[b, size -1];
       }
```

4/ أكتب إعلاناً لمصفوفة أعداد صحيحة تدعى intArray والتي تحتوي على ثلاثة صفوف وعمودين.

5/ أكتب برنامجاً يقوم بطباعة العنصر الأصغر من عناصر مصفوفة تحتوي على ثلاثة صفوف وثلاثة أعمدة.

6/ أكتب عبارة C++ صحيحة لكل من الآتي (إفترض أنه تم الإعلان عن عددين صحيحين Value1 و Value2 وتم تمهيد قيمة المتغير value1 عند 200000) :

- الإعلان عن مؤشر iptr ليشير إلى متغير من النوع int.
- تعيين عنوان المتغير value1 إلى المؤشر iptr.
- طباعة القيمة التي يشير إليها المؤشر iptr.
- تعيين القيمة التي يشير إليها المؤشر iptr إلى المتغير value2.
- طباعة قيمة المتغير value2.
- طباعة عنوان المتغير value1.
- طباعة العنوان المخزن في المؤشر iptr. (هل تتساوى هذه القيمة مع عنوان المتغير value1؟)

6.0

الوحدة السادسة

Classes (I) – الفئات(I)



بنهاية هذه الوحدة:

- ستتعرف على كيفية إنشاء الفئات في لغة ***C++***.
- ستتعرف على كيفية إنشاء واستعمال كائنات الفئات.
- ستتعرف على كيفية الوصول إلى الأعضاء البيانية والدالية في الفئة.
- ستتعرف على مفهوم البنيات في لغة ***C++***.

6.1 مقدمة

أساس البرامج المكتوبة باللغة **C++** هو الكائنات التي يتم إنشاؤها بواسطة فئة تستعمل كقالب فعندما يكون هنالك الكثير من الكائنات المتطابقة في البرنامج لا يكون منطقياً وصف كل واحد منها على حدة ، من الأفضل تطوير مواصفات واحدة لكل من هذه الكائنات وبعد تحديد تلك المواصفات يمكن استخدامها لإنشاء قدر ما نحتاج إليه من الكائنات تسمى مواصفات إنشاء الكائنات هذه في *OOP* فئة **(Class)** . تتميز الفئة في **C++** بالملامح الأربعة التالية :-

1/اسم الفئة والذي يعمل كنوع البيانات الذي ستمثله الفئة.

2/مجموعة من الأعضاء البيانية في الفئة **(data members)** حيث يمكن أن تحتوى الفئة على صفر أو أكثر من أي نوع من أنواع البيانات في **C++** .

3/مجموعة من الأعضاء الدالية **(member functions)** معرفة داخل الفئة وهي تمثل مجموعة العمليات التي سيتم تنفيذها على كائنات الفئة.

4/ محددات وصول **(access specifiers)** وتكتب قبل الأعضاء البيانية والأعضاء الدالية لتحدد إمكانية الوصول إلى هذه الأجزاء من الأجزاء الأخرى في البرنامج.

6.2 تعريف الفئة The Class Definition

يتألف تعريف الفئة من الكلمة الأساسية **class** يليها اسم الفئة ثم جسم الفئة بين قوسين حاصرين { } ويجب أن ينهي تعريف الفئة فاصلة منقوطة أو عبارة إعلان عن كائنات تنتمي إلى الفئة فمثلاً:

```
class anyclass {  /* class body*/  };
```

أو

```
class anyclass { /* class body */ }  obj1, obj2;
```

غالباً ما تكتب الفئة في **C++** على النحو التالي في البرنامج :

```
class class_name{
private:
   data members
public:
   member functions
};
```

المثال التالي يوضح كيفية تعريف فئة تدعى **stack** :-

```
// This creates the class stack >
class stack   {
private:
    int stck[SIZE];
    int tos;
public:
    void  init (   );
       void push(int i);
       int pop (  );
};
```



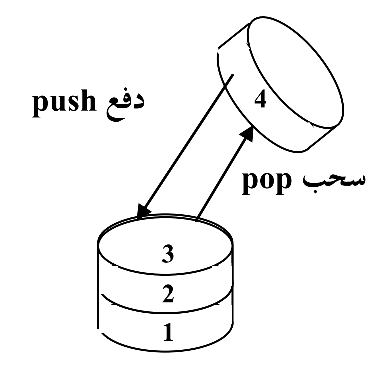
كما عرفنا سابقاً أن المصفوفة هي طريقة لتخزين البيانات ولكنها غير مناسبة في الكثير من الحالات .

يمكن إنشاء بنيات تخزين أخرى كاللوائح المرتبطة **(linked lists)** والمكدسات **(stacks)** والصفوف **(queues)** . كل من بنيات التخزين هذه لها حسناتها ومساوئها وتختلف فيها الطريقة التي يتم استخدامها للوصول إلى البيانات المخزنة فيها .

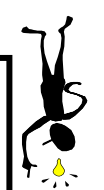
المكدس **(stack)** هو نوع من بنيات التخزين يستخدم عندما نريد الوصول إلى آخر عنصر تم تخزينه . يشار إلى هذه البنية عادة **lifo** اختصاراً لـ **last in first out** والتي تعنى (المدخل آخراً هو المخرج أولاً).

تستطيع المكدسات **(stacks)** تخزين أي نوع من البيانات . لكن كما هو الحال مع المصفوفات يخزن كل مكدس نوعاً واحداً من البيانات ، ولكن ليس خليطاً من الأنواع.

عندما نضع قيمة في المكدس ، يقال أننا دفعناها **(push)** في المكدس ، وعندما نخرج القيمة منه يقال أننا سحبناها **(pop)**. يبين الشكل التالي كيف يبدو هذا:



تستعمل عادة لتمثيل المكدس مصفوفة يتم فيها تخزين البيانات ، ومؤشر يشير إلى أعلى المكدس (آخر عنصر في المكدس) .



إن مواصفات الفئة لا تؤدى إلى إنشاء أي كائن **stack**، بل ستقوم فقط بتحديد كيف سيبدو الكائن عند إنشاءه.

داخل جسم الفئة يتم الإعلان عن الأعضاء البيانية والأعضاء الدالية ومحددات الوصول لها وفيما يلي سنتعرف على هذه الأجزاء .

6.3 الأعضاء البيانية Data Members

يتم الإعلان عن الأعضاء البيانية في الفئة بنفس الطريقة التي يتم بها الإعلان عن المتغيرات باستثناء أنه لا يمكننا تمهيد الأعضاء البيانية عند الإعلان عنها، يمكن أن تكون الأعضاء البيانية من أي نوع بيانات في الـ C++ فمثلاً في الفئة **stack** تم الإعلان عن الأعضاء البيانية كما يلي :

```
int stck[SIZE];
int tos;
```

تحتوى الفئة **stack** على بندى بيانات هما مصفوفة **stck** عناصرها من النوع **int** ومتغير **tos** من النوع **int** أيضاً . لاحظ أن هذه التعريفات لا تعطى المتغيرات أي قيمة هي فقط تعطيها اسماً وتحدد أنها تتطلب مساحة معينة من الذاكرة حيث يتم تخصيص مساحة الذاكرة بعد إنشاء الكائنات.

6.4 الأعضاء الدالية Member Functions

يمكن لمستخدمي الفئة **stack** إنجاز العديد من العمليات على الكائنات التابعة لها ، يتم الإعلان عن هذه العمليات داخل جسم الفئة ويطلق عليها الأعضاء الدالية أو: (member functions) ويتم تصريحها داخل جسم الفئة ، فمثلاً في الفئة **stack** تم تصريح الأعضاء الدالية كالآتي :

```
void  init (   );
void push (int i);
int pop (  );
```

هنالك ثلاث دالات في مواصفات الفئة stack ، Pop( ) وPush( ) و int( ). لا تعيد الدوال int( ) ، Push( ) أي قيمة بينما تعيد الدالة Pop( ) قيمة من النوع int . تسمى الدوال المعرفة داخل الفئة أعضاء دالية member functions .

6.5 محددات الوصول Access Specifiers

يتم تحديد إمكانية الوصول إلى أعضاء الفئة (بيانات ، أعضاء دالية) باستخدام ثلاث كلمات أساسية في C++ وهي public (عام) و private (خاص) وprotected (محمي) والتي تتم كتابتها داخل جسم الفئة تليها نقطتان(:).

- العضو العام publicفي الفئة يمكن الوصول إليه من أي مكان داخل البرنامج.
- العضو المحمى protected في الفئة يمكن الوصول إليه فقط من فئته أو الفئات المشتقة منها كما سنرى لاحقاً .
- العضو الخاص private يمكن الوصول إليه فقط من الأعضاء الدالية في فئته والفئات الصديقة لها كما سنرى لاحقاً.

إذا لم يتم ذكر محدد وصول لعضو في فئة ما سيفترض المصرف أن محدد الوصول لهذا العضو هو private.

في الفئة stack كل البيانات خاصة وكل الأعضاء الدالية عامة وهذه هي الحالة العامة فى C++ لأننا نريد أن نخفى البيانات عن العالم الخارجي لا يمكن أن تكون محمية بينما نريد أن تكون الأعضاء الدالية عامة حتى تستطيع الأجزاء الأخرى من البرنامج استدعائها.

6.6 إنشاء الكائنات والتفاعل معها

عرفنا أن الهدف الأساسي من الفئة هو استعمالها كأساس لإنشاء الكائنات . ولكن كيف يتم إنشاء الكائنات ؟

يمكن إنشاء الكائنات باستعمال نفس التركيب المستخدم لإنشاء متغير من نوع أساسي كـ int مثلاً وذلك أن الكائنات في C++ تتم معاملتها كأنواع متغيرات كما تتم معاملة الفئات كأنواع بيانات وعليه لإنشاء كائن تابع للفئة stack نكتب:-

```
stack stack1;
```

عند تنفيذ العبارة يحسب البرنامج حجم الكائن ويخصص مساحة كافية له من الذاكرة ويعطى مساحة الذاكرة هذه اسماً stack1 . وبنفس الطريقة يمكننا إنشاء قدر ما نشاء من الكائنات :-

```
stack stack1, stack2 ,stack3;
```

التفاعل مع الكائنات:-

يتم التفاعل مع الكائنات من خلال استدعاء أحد أعضاءها الدالية والثاني يبدو كإرسال رسالة إلى الكائن. نحتاج إلى تركيب مؤلف من قسمين : اسم الكائن واسم العضو الدالي ويتم ربط اسم الكائن واسم الدالة بواسطة نقطة(.) تسمى عامل الوصول إلى أعضاء الفئة.

عامل دقة المدى:- scope resolution operator

يتم تصريح الأعضاء الدالية داخل جسم الفئة ولكن قد تحتاج إلى تعريف أحد الأعضاء الدالية خارج جسم الفئة، عندها يجب أن يتضمن اسمه اسم الفئة التي يتبع لها وإلا لن تكون هنالك طريقة لكي يتمكن المصرف من معرفة الفئة التي ينتمي إليها العضو الدالي . يتم ربط اسم الدالة مع اسم الفئة باستعمال ما يسمى بعامل دقة المدى. يتألف هذا العامل من نقطتين مزدوجتين :: ، المثال التالي يوضح تعريف الدالة Push التي تنتمي إلى الفئة stack .

```
void stack::push(int  i)
{
    if(tos==SIZE)  {
      cout << "stack is full.\n";
      return;
     }
     stck[tos] = i;
     tos++;
}
```

البرنامج التالي يوضح كيفية استخدام الفئة **stack** التي قمنا بتعريفها.

```
//Program 6-1:
#include<iostream.h>
const int SIZE= 100;
// This creates the class stack.
//Continued
class stack   {
private:
     int stck[SIZE];
     int tos;
public:
    void  init (   );
    void push (int i);
    int pop (  );
};

void stack:: init ( )
{
   tos = 0;
}
void stack::push (int i)
{
if (tos == SIZE )   {
        cout << "Stack is full.\n";
        return;
   }
   stck[ tos] = I;
   tos++;
}

int stack::pop( )
{
```

```
    if(tos == 0)  {

      cout << "Stack underflow.\n" ;
      return 0;
}
tos--;
   return stck[tos];
}
//Continued
int main ( )
{
   stack  stack1, stack2;   // create two stack objects

   stack1.init ( );
   stack2.init ( );

   stack1.push (1);

  stack2.push (2);

   stack1.push (3);
   stack2.push (4);

    cout << stack1.pop( ) << "    ";
    cout << stack1.pop( ) << "    ";
    cout << stack2.pop( ) << "    ";
    cout << stack2.pop( ) << "\n";

   return 0;
}
```

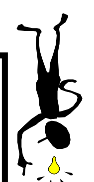
عندما نسحب البيانات التي قمنا بدفعها في المكدس تظهر بترتيب معكوس وعليه الخرج من البرنامج :

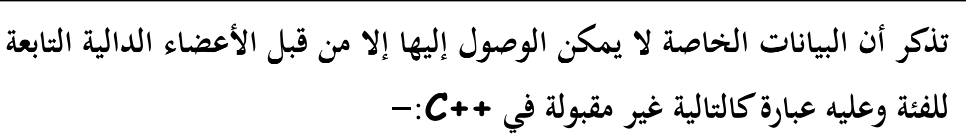
```
3  1  4  2
```

لاحظ أننا استعملنا العاملين المتصدر **(++tos)** واللاحق **(--tos)** لمعالجة فهرس المصفوفة **stck** . يمثل المتغير **tos** أعلى المكدس وقد تم تمهيده عند **0** .

عند دفع البيانات في المكدس تتم زيادة **tos** أولاً ثم يتم استعماله كفهرس لذا نجد أن **tos** يشير دائماً إلى مكان واحد قبل بند البيانات الأخير المدفوع في المكدس.

عند سحب البيانات يتم الوصول إليها أولاً ثم يتم إنقاص الفهرس **(--tos)** لذا فإن **tos** يشير مباشرة إلى أعلى المكدس.





تذكر أن البيانات الخاصة لا يمكن الوصول إليها إلا من قبل الأعضاء الدالية التابعة للفئة وعليه عبارة كالتالية غير مقبولة في ++C:-



```
stack1.tos=0 // Error tos is private
```

كيفية الوصول إلى الأعضاء العامة في الفئة:

للوصول إلى الأعضاء العامة في فئة ما، يمكن استخدام:

١ - إسم كائن تابع للفئة وعامل النقطة **(.)** .

٢ - مرجع إلى كائن في الفئة وعامل النقطة.

٣ - مؤشر إلى كائن في الفئة والعامل **(->)**.

البرنامج التالي يوضح هذا:

```
//Program 6-2:
#include<iostream.h>
class count {
public:
  int x;
```

```
  void print( ) { cout <<x<<endl;}
};
main( )
{
  count counter;
//Continued
*countrptr=&counter;
cout<<"assign 7 to x and pring using the object's name: ";
counter.x=z;
counter.print( );
cout<<"assign 8 to x and print using a reference: ";
countref-x=9;
cout <<countref.print( );
cout<<"assign 10 to x and print using a pointer: ";
counterptr->x=10;
counterptr->print( );
return 0;
```

6.7 البنيات
structures

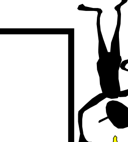
البنية في **C++** هي طريقة لتجميع عدة بنود بيانات يمكن أن تكون من أنواع مختلفة .

يتم استعمال البنيات عادة عندما تشكل عدة بنود بيانات وحدة متميزة لكنها غير مهمة لتصبح فئة . وعلى الرغم من أن الفئة في **C++** تنفذ كل المهام التي تقوم بها البنيات لكن لا يزال هنالك الكثير من الحالات التي تكون فيها البنيات مفيدة . وكمثال على بنية:-

```
struct part
{
int modelnumber;
int partnumber;
float cost;
};
```

تتألف البنية من الكلمة الأساسية **struct** يليها اسم البنية وأقواس حاصرة تحيط بجسم البنية. تنهى البنية فاصلة منقوطة.

يتألف جسم البنية عادة من عدة بنود بيانات يمكن أن تكون من أنواع مختلفة تسمى هذه البنود أعضاء **members** .



- البنية شبيهة جداً بالفئة من ناحية التركيب المنطقي لكن الفئات والبنيات تستعمل بطرق مختلفة جداً ، عادة تحتوى الفئة على بيانات ودالات بينما تحتوى البنية على بيانات فقط.
- لا تؤدى مواصفات البنية إلى إنشاء أي بنية كما هو الحال مع الفئات ،إنها مجرد مواصفات لشكل البنية عندما يتم إنشاؤها.

لتعريف متغيرات من النوع البنيوي **part** نكتب:

```
part cp1,cp2;
```

هنالك أيضاً طريقة مختصرة لتعريف المتغيرات البنيوية حيث يتم وضع أسماء المتغيرات في مواصفات البنية كالآتي:

```
struct part
{
```

```
int modelnumber;
int partnumber;
float cost;
}cp1,cp2;
```

6.8 الوصول إلى أعضاء البنية Accessing structs

يتم استعمال عامل النقطة للوصول إلى أعضاء البنية تماماً مثلما يتم استعماله للوصول إلى الأعضاء الدالية من الكائنات ،فمثلاً يمكننا أن نكتب:-

```
cin>> cp1.part number;
```

ويكون اسم المتغير قبل النقطة بينما يكون اسم العضو البياني بعدها.

تمهيد المتغيرات البنيوية:

يمكن تزويد قيم أولية للمتغيرات البنيوية تماماً كما نفعل مع المصفوفات ، فمثلاً لتمهيد متغير بنيوي من النوع part نكتب:

```
part cp1 = {6244,15,217.1};
```

تؤدى هذه العبارة إلى تمهيد cp1.modelnumber عند القيمة 6244 و cp1.partnumber عند القيمة 15 وتمهيد cp1.cost عند القيمة 217.1 .

إستعمال البنية:

في الفئة stack التي قمنا بتعريفها في الأمثلة السابقة نجد أن المصفوفة التي يتم فيها تخزين بنود البيانات والمتغير tos الذي يشير إلى أعلى المكدس stack مرتبطان ببعضهما إلى حد كبير لذلك من الأنسب دمجهما في بنية ، ويتم استعمال هذه البنية كعضو بياني واحد في الفئة stack فيما يلي سنوضح كيف يكون هذا:

```
//Program 6-3:
# include<iostream.h>
# define size 100
```

```
sruct stackette
//Continued
{
int stck[size];
int tos;
};
class stack
{
private:
stackette st;
public:
void init( );
void push( int i);
int pop( );
};
void stack :: init( )
{
st.tos=0;
}
void stack:: push(int i );
{
if(st.tos== size){
cout <<"stack is full.\n";
return;
}
st.stck[st.tos] = i;
st.tos ++;
}
int stack:: pop( )
{
if(st.tos== 0) {
cout <<"stack under flow.\n";
return 0;
}
```

```
st.tos--;
return st.stck[st.tos];
//Continued
}
int main( )
{
stack stack1;
stack1.init( );
stack1.push(1);
stack1.push(2);
stack1.push(10);
cout<< stack1.pop( )<< " " ;

cout<< stack1.pop( )<< " " ;
return 0;
```

الخرج من هذا البرنامج :

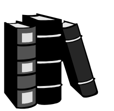
```
2
```

تخزن البنية **stackette** هنا مصفوفة أعداد صحيحة ومتغير يشير إلى أعلى المكدس. العضو البياني الوحيد في الفئة **stack** الآن هو متغير تابع للبنية **stackette** وتشير الأعضاء الدالية للفئة **stack** الآن إلى الأعضاء البيانية في **st** باستعمال عامل النقطة st.tos=0

6.9 البنيات مقابل الفئات Structs vs. Classes

يمكن القول أن البنية هي تجميع هامد لبنود البنيات بينما الفئة هي آلية نشطة للبيانات والدالات ، فالفئات تشكل أساس البرمجة الكائنية المنحى بينما البنيات هي جزء صغير في استعمالات **C++** . نجد أن التركيب المنطقي للفئات والبنيات متطابق تقريباً ،إلا أن أعضاء

الفئة تكون أعضاء خاصة بشكل افتراضي . أي إذا لم يتم استعمال الكلمات الأساسية public أو private تكون أعضاء الفئة خاصة.



الملخص:

- أساس البرامج المكتوبة باللغة **C++** هو الكائنات.
- تأخذ الفئة في **C++** الشكل العام التالي:

```
class   classname {
```

- تحتوى الفئة على بيانات معرفة داخله وتسمى أعضاء بيانية **(data members)** وعلى دالات تسمى أعضاء دالية **(function members)** .
- يتم إنشاء الكائنات باستعمال نفس التركيب المستخدم لإنشاء متغير من نوع أساسي .
- تعامل الكائنات في **C++** كأنواع متغيرات كما تتم معاملة الفئات كأنواع بيانات.
- لإنشاء كائن **anyobj** تابع للفئة **anyclass** نكتب:

```
anyclass   anyobj;
```

- يتم التفاعل مع الكائنات باستدعاء أحد أعضائها الدالية والذي يبدو كإرسال رسالة إلى الكائن.
- للتفاعل مع الكائنات تتم كتابة اسم الكائن واسم العضو الدالي ويتم ربط اسميهما بواسطة نقطة (.) تسمى عامل الوصول إلى أعضاء الفئة.
- إذا تم تعريف عضو دالي خارج فئته يتم ربط اسم فئته بواسطة العامل (::) والذي يسمى بعامل دقة المدى.
- البيانات الخاصة لا يمكن الوصول إليها إلا من قبل الأعضاء الدالية التابعة للفئة.
- البنية في **C++** هي طريقة لتجميع عدة بنود بيانات يمكن أن تكون من أنواع مختلفة.
- يتم استعمال البنيات عندما تشكل عدة بنود بيانات وحدة متميزة لكنها غير مهمة لتصبح فئة.
- تتألف البنية من الكلمة الأساسية **struct** يليها اسم البنية وأقواس حاصرة تحيط بجسم البنية وتنهى البنية فاصلة منقوطة.
- يتألف جسم البنية من عدة بنود بيانات يمكن أن تكون من أنواع مختلفة وتسمى تلك البنود أعضاء.
- يتم استعمال عامل النقطة للوصول إلى أعضاء البنية تماماً مثلما يتم استعماله للوصول إلى الأعضاء الدالية من الكائنات.



الأسئلة

1/ أنشئ فئة تدعى **complex** تقوم بإجراء العمليات الحسابية على الأعداد المركبة.

العدد المركب يكون على الصورة :

real part + imaginary part*i

حيث $i=\sqrt{-1}$

استخدم متغيرات من النوع float لتمثيل البيانات الخاصة في الفئة، على أن تحتوى الفئة complexعلى الدوال الآتية:

- **دالة تقوم بجمع عددين مركبين.**
- **دالة تقوم بطرح عددين مركبين.**
- **دالة تقوم بطباعة الأعداد المركبة على الصورة (a, b) حيث a يمثل الجزء الحقيقي ، b تمثل الجزء التخيلي.**

قم بكتابة برنامج ++C كاملاً لاختبار الفئة التي قمت بإنشائها.

2/ أنشئ فئة تدعى **Rational** والتي تجرى العمليات الحسابية على الكسور **fractions**.

استخدم متغيرات من النوع int لتمثيل البيانات الخاصة في الفئة(البسط والمقام).

تحتوى الفئة Rational على دوال تقوم بالعمليات الآتية:-

- **جمع عددين من النوع Rational.**
- **طرح عددين من النوع Rational.**
- **ضرب عددين من النوع Rational.**
- **قسمة عددين من النوع Rational.**
- **طباعة الكسور على الصورة a/b حيث يمثل a البسط و b المقام.**

3/ **أوجد الخطأ في الآتي:-**

البرنامج التالي هو جزء من تعريف فئة تدعى Time:

```
class Time {
public:
```

```
        // function prototypes
private:
int hour = 0;
int minute = 0;
int second = 0;
};
```

4/ ما هو الغرض من عامل دقة المدى ::. scope resolution operator

5/ قارن بين مفهومي البنيات والفئات في *C++*.

الوحدة السابعة Classes (II) – (II) الفئات	7.0



بنهاية هذه الوحدة:

- ستتعرف على المشيدات constructors.
- ستتعرف على المهدمات destructors .
- ستتمكن من إنشاء كائنات ثابتة Constant objects وأعضاء دالية ثابتة Constant member functions.
- **ستتمكن من استعمال أعضاء بيانية ساكنة** Static data members وأعضاء دالية ساكنة Static member functions.

7.1 المشيدات

Constructors

في بعض الأحيان نحتاج لتمهيد الكائنات عند قيم معينة قبل استعمالها في البرنامج ،فمثلاً في الفئة **stack** والتي تم تعريفها سابقاً المتغير **tos** تم تمهيد قيمته عند 0 وذلك باستعمال الدالة ( )int .

إن تمهيد المتغير **tos** عند 0 باستعمال دالة كـ ( )int مثلاً ليس أسلوباً مفضلاً في OOP ، أحد أسباب هذا أن المبرمج الذي يكتب الدالة ( )main يجب أن يتذكر ضرورة استدعاء هذه الدالة كلما تم استدعاء كائن تابع للفئة **stack** ، لذلك تسمح ++C للكائنات بتمهيد نفسها عند إنشائها هذا التمهيد يتم استعمال دوال خاصة تسمى المشيدات.

المشيد: هو عضو دالي خاص يحمل نفس اسم الفئة ويتم استعماله لتمهيد الكائنات .

النموذج التالي يوضح كيف تبدو فئة **stack** عند استعمال مشيد لتمهيد المتغير **tos**.

```
//Program 7-1:
// This creates the class stack.
const int SIZE= 100;
class stack {
 int stck[size];
 int tos;
public:
 stack( );      //constructor
  void push (int i);
 int pop( );
};
```

لاحظ أن المشيد **stack** لا يحمل أي قيمة إعادة. في ++C لا ترجع المشيدات أي قيم عند استدعائها هي فقط تقوم بتمهيد الكائنات عند قيم معينة.

7.2 المهدمات
Destructors

إن كل كائن يتم إنشاؤه سيتم تدميره في وقت ما لذا في **C++** بإمكان كاتب الفئة كتابة مهدم بنفسه، يعمل هذا المهدم على إلغاء تخصيص الذاكرة التي كان المهدم قد خصصها للكائن . يحمل المهدم أيضاً نفس اسم الفئة لكن تسبقه العلامة **~** .أيضاً لا يملك المهدم قيمة إعادة.

لنرى كيفية عمل دوال المشيدات والمهدمات . المثال البرنامج يوضح إصدار جديد من الفئة stack

---

```
//Program 7-2:
// using a constructor and destructor.
#include<iostream.h>
const int SIZE=100;
//This creates the class stack.
class stack {
int stck[SIZE];
int tos;
public:
stack( );  // constructor
~stack( );    //destructor
void push(int i);
int pop( );
};
// stack's constructor function
stack::stack( )
{
tos=0;
cout<<"stack Initialized\n";
}
// stack's destructor function
stack::~stack( )
```

```
{
cout << "stack Destroyed\n";
//Continued
}
void stack :: push(int i)
{
if(tos == SIZE)      {
cout << "stack is full.\n";
return;
}
stack[tos] = i;
tos++;
}
int stack::pop( )
{
if(tos== 0) {
cout<<"stack underflow.\n";
return 0;
}
tos--;
return stck[tos];
}
int main( )
{
stack a, b; // create two stack objects
a.push(1);
b.push(2);
a.push(3);
b.push(4);
cout <<a.pop( )<<" ";
cout <<a.pop( )<<" ";
cout <<b.pop( )<<" ";
cout <<b.pop( )<<"\n ";
return 0;
```

```
}
```

الخرج من البرنامج

```
Stack Initialized
Stack Initialized
3  1  4  2
Stack Destroyed
Stack Destroyed
```

وسائط المشيدات Parameterized constructor:-

المشيدات التي لا تأخذ وسيطات كالمشيد المستخدم في الفئة stack تسمى مشيدات اشتقاق ولكن من الممكن تمرير وسائط إلى المشيدات بنفس الطريقة التي تمرر بها إلى الدوال الأخرى.

المثال البرنامج يحتوى على مشيد مع وسيطات.

```
//Program 7-3:
#include <iostream.h>
class myclass {
int a, b;
public:
myclass(int i,int j)    {a=i; b=j;}
void show ( ) {cout <<a<<" " <<b;}
};
int main( )
{
myclass ob(3, 5);
ob.show( );
return 0;
}
```

لاحظ في تعريف المشيد ( ) myclass تم تمرير وسيطتين هما i و j واستعملت هاتين الوسيطتين لتمهيد القيم a و b .

يوضح المثال كيفية تمرير الوسائط عند إنشاء الكائن فالعبارة :-

myclass ob(3,4);

تتسبب في إنشاء كائن يدعى ob وتقوم بتمرير القيم 3 و 4 كوسائط. يمكننا أيضاً تمرير قيم الوسائط باستعمال العبارة التالية:

myclass ob= myclass (3,4);

ولكن العبارة الأولى هي الأكثر استخداماً .

المشيد أحادى الوسيطات :- Constructor with one parameter

في المشيد أحادى الوسيطات هنالك طريقة ثالثة لتمرير الوسيطة إليه. المثال التالي يوضح كيف يكون هذا:

```
//Program 7-4:
#include<iostream.h>
class X {
int a;
public:
X(int j) {a= j;}
Int geta( ) {return a; }
};
int main( )
{
X ob = 99;  //passes 99 to j
cout<<ob.geta( ); // outputs 99
return 0;
}
```

هنا المشيد x يأخذ وسيطة واحدة . لاحظ الطريقة التي تم بها تعريف الكائن ob داخل الدالة ( )main . تم تمهيد قيمة وسيطة المشيد x عند 99 وذلك بكتابة :-

x ob= 99

وعموماً إذا كنا نتعامل مع مشيد ذو وسيطة واحدة يمكننا تمرير الوسيطة إما بكتابة ob(i) أو ob=i.

يلعب المشيد أحادى الوسيطات دوراً مميزاً في البرمجة كائنية المنحى حيث يمكن استعماله لتحويل كائن منحى من فئة إلى فئة أخرى وذلك بتمرير الكائن كوسيطة للمشيد يطلق على هذه مشيدات دالة تحويل.

متى يتم تنفيذ المشيدات والمهدمات :-

يتم استدعاء المشيدات كلما تم إنشاء كائن ، ويتم استدعاء المهدم لكل كائن قبل تدميره ، ولمعرفة متى يتم تنفيذ المشيدات والمهدمات أدرس البرنامج :

```
//Program 7-5:
#include<iostream.h>
class myclass  {
public:
int who;
myclass(int id);
~myclass( );
} glob_ob1(1), glob_ob2(2);

myclass::myclass(int id)
{
cout<<"Initializing"<<id<<"\n";
who = id
}
myclass::~myclass( )
//Continued
{
cout<<"Destructing"<<who<<"\n";
}
int main( )
{
myclass local_ob1(3);
cout <<"this will not be first line displayed.\n";
```

```
myclass local_ob2(4);
return 0;
}
```

الخرج من البرنامج:

```
Initializing 1
Initializing 2
Initializing 3
This will not be first line displayed.
Initializing 4
Destructing4
Destructing3
Destructing2
Destructing1
```

كما رأينا في البرنامج السابق الكائنات المعرفة داخل الدالة **main( )** يتم تنفيذ مشيداتها بترتيب إنشاء الكائنات بينما يتم تنفيذ مهدماتها بعكس ترتيب إنشاء الكائنات وعليه يتم تنفيذ مشيد الكائن **local ob 1** يليه الكائن **local ob 2** بينما يتم تنفيذ مهدم الكائن **local ob 2** قبل مهدم الكائن **local ob 1** .

يتم تنفيذ مشيدات الكائنات المعرفة داخل الفئة قبل تنفيذ الدالة **main ( )** وأيضاً يتم تنفيذ مهدماتها بترتيب معكوس ولكن بعد نهاية تنفيذ الدالة **main( )** .

لنبرهن على مدى تنوع استعمالات فئات لغة **C++** سنقوم في البرنامج التالي بتعريف فئة لشيء مختلف : نوع بيانات جديد يمثل الوقت **(Time)** ، يتألف هذا الوقت من ثلاث بيانات الساعات، الدقائق والثواني، وسنسمى نوع البيانات الجديد هذا **Time**

```
//Program 7-6:
// Time class.
```

```
#include<iostream.h>
// Time abstract data type (ADT) definition
class Time  {
public:
//Continued
Time( );
    void setTime (int, int, int)
    void printMilitery( );
void printStandard( );
private:
   int hour;
   int minute;
   int second;
};
Time::Time ( )   { hour = minute = second = 0;   }
void Time::setTime (int h, int m, int s)
{
    hour = (h >= 0 && h < 24) ? h : 0;
    minute = (m >= 0 && m < 60) ? m : 0;
    second = (s >= 0 && s < 60) ? s : 0;
}
void Time::printMilitary( )
{
cout << (hour < 10 ? "0" : " " ) << hour << ":"
      << (minute < 10 ? "0" : " ") << minute << ":"
      << (second < 10 ? "0" : " " )<< second;
}
void Time::printStandard( )
{
cout<< ((hour ==0 | | hour == 12 )? 12  : hour % 12)
      <<  ":" <<(minute < 10 ? "0" : " ") << minute
      << ":" <<(second < 10 ? "0" : " " )<< second
      << (hour < 12 ? " AM"  :   "PM");
}
```

```
int main ( )
{
        Time t;
        cout<< "The initial military time is: ";
       t.printMilitary( );
  //Continued
   cout<< endl <<"The initial standard time is: ";
   t.printStandard( );
t.setTime(13, 27, 6) ;
   cout<< endl << endl << "Military time after setTime is  ";
   t.printMilitary( );
   cout<< endl << "Standard time after setTime is  ";
   t.printStandard( );
  return 0;
}
```

الخرج من البرنامج:

```
The initial military time is 00:00:00
The initial standard time is 12:00:00 AM

Military time after setTime is 13:27:06
Standard time after setTime is 1:27:06 PM
```

ينشئ البرنامج كائناً واحداً تابع للفئة Time هو t. عندما يتم إنشاء الكائن t يتم استدعاء المشيد Time والذي يقوم بتمهيد بيانات الكائن t عند 0 . يتم طباعة قيمة الكائن t باستخدام تنسيقين :

- Standard: والذي يستعمل التنسيق 24-ساعة.
- Military: والذي يستعمل التنسيق 12-ساعة.

ثم يستعمل الدالة **setTime** وطباعة الكائن **t** مرة أخرى باستخدام التنسيقين.

7.3 الكائنات الثابتة Constant Objects

لقد رأينا كيف يمكن استعمال متغيرات ثابتة ذات أنواع أساسية ، حيث تم استعمالها لتعريف ثابت كحجم مصفوفة ، يمكن جعل كائن تابع لفئة ما ثابتاً إذا كنا نريد ضمان عدم تغير البيانات في الكائن وكمثال على ذلك في الفئة **Time** والتي رأيناها في البرنامج السابق، لنفترض أننا نريد إنشاء كائن يدعى **noon (12, 0, 0)** سيكون من الجيد ضمان عدم تغيير قيمة هذا الكائن لتحقيق هذا نكتب العبارة

```
const Time noon( 12, 0, 0);
```

والتي تعلن عن كائن ثابت **noon** في الفئة **Time** وتمهد قيمته عند **12** .

لا تسمح مصرفات ++C باستدعاء الكائنات الثابتة من قبل الأعضاء الدالية في الفئة لضمان عدم تعديل بيانات هذه الكائنات ، ولكن قد نرغب في بعض الأحيان في عرض قيمة هذه الكائنات والتي لا تؤثر بأي حال من الأحوال على بياناتها ، لحل هذه المشكلة يمكن للمبرمج الإعلان عن دالات ثابتة **(const)** وهي عبارة عن أعضاء دالية تضمن أنه لن يتم تغيير بيانات الكائن الذي استدعي من أجلها ، ولجعل عضو دالي ثابتاً تتم كتابة الكلمة الأساسية **const** في تعريف العضو الدالي وتصريحه مباشرة بعد الأقواس التي تلي اسمه .

أدناه يبدو العضو الدالي **printMilitary** التابع للفئة **Time** :-

```
void Time::printMilitary( ) const
{
cout<< (hour < 10 ? "0" : " " ) << hour << ":"
      << (minute < 10 ? "0" : " ") << minute << ":"
      << (second < 10 ? "0" : " " )<< second;
}
```

البرنامج التالي يوضح استخدام الكائنات والأعضاء الدالية الثابتة:

```
//Program 7-7:
class Time {
public:
Time( );
```

```
void setTime ( int, int, int);
void printMilitary( ) const;
void printStandard( )const;
private:
int hour;
int minute;
int second;
};
void Time:: setTime (int h, int m, int s)
{
//Continued
hour = (h >=0 && h<24) ? h : 0;
minute = (m >= 0 && m<60 ) ? m : 0;
second = (s >= 0 && s<60) ? s : 0;
}
void Time::printMilitary( ) const
{
cout << (hour < 10 ? "0" : " " ) << hour << ":"
       << (minute < 10 ? "0" : " ") << minute << ":"
       << (second < 10 ? "0" : " " )<< second;
}

void Time::printStandard( ) const
{
cout << ((hour ==0 | | hour == 12 )? 12  : hour % 12)
       <<  ":" <<(minute < 10 ? "0" : " ") << minute
       << ":" <<(second < 10 ? "0" : " " )<< second
       << (hour < 12 ? " AM"  :   "PM");
}
int main ( )
{
const Time noon(12, 0, 0);
cout <<" noon = " ;
noon.printStandard;
```

```
Time t;
t.setTime (13, 27, 6);
cout<< endl << "military time after setTime is ";
t.printMilitary ( );
cout<< endl;
return 0;
}
```

الخرج من البرنامج:

```
noon = 12:00:00 AM
military time after setTime is 13:27:06
```

في البرنامج السابق تم تعريف كائنين في الفئة Time أحدهما ثابت هو noon على عكس الآخر وهو t. العضوان الداليان printStandard( ) و printMilitary( ) ثابتان لا يعدلان كائنهما لكن العضو الدالي setTime يعدل كائنه لذا لم يجعل ثابتاً. يمكننا استدعاء setTime( ) للكائن t لكن ليس للكائن noon.

7.4 الأعضاء الساكنة في الفئات Static class member

(أ) البيانات الساكنة :-

استعملنا حتى الآن أعضاء بيانية مثيلية (instant) أي أن هنالك نسخة واحدة منها لكل كائن يتم إنشاؤه ولكن قد نحتاج لمتغير ينطبق على كل كائنات الفئة ، لتحقيق ذلك نستعمل عضواً بيانياً ساكناً static data member فعندما نعلن عن متغير في بيانات فئة ما على أنه ساكن static نعنى بذلك أنه ستكون هنالك نسخة واحدة فقط من هذا المتغير في الذاكرة وستتشارك كل الكائنات التابعة لهذه الفئة في هذا المتغير بغض النظر عن عدد هذه الكائنات . يتم تمهيد كل المتغيرات الساكنة عند 0 قبل إنشاء أي كائن .

يتم تصريح المتغير الساكن ضمن الفئة باستعمال الكلمة الأساسية **static** ويتم تعريفه خارجها ، وذلك لأنه إذا افترضنا أن البيانات المعرفة داخل الفئة هي بيانات مثيلية مكررة لكل كائن ، إذن لتعريف متغير يتواجد مرة لكل الفئة علينا تعريفه خارج الفئة وتصريحه داخلها ليكون معروفاً لبقية أعضائها.

لتوضيح استعمال وتأثير البيانات الساكنة ادرس المثال البرنامج:

```
//Program 7-8:
#include<iostream.h>
class shared  {
static  int a;
int b;
//Continued
public:
void set(int i,int j) { a=i; b=j;}
void show( );
};
int shared :: a; // define a
void shared :: show( )
{
cout <<" This is static a:   "<< a;
cout<<"\nThis is non_static b:  " <<  b;
cout << "\n";
}
int main( )
{
shared x, y;
x.set(1, 1);     //set a to 1
x.show( );
y.set(2, 2);     //change a to 1
y.show( );
x.show( );    /* Here, a has been changed for both x and y
                  because a is shared by both objects.*/
return 0;
```

```
}
```

الخرج من البرنامج:

```
This is static a:  1
This is non_static b:  1
This is static a:  2
This is non_static b:  2
This is static a:  2
This is non_static b:  1
```

(ب) **الأعضاء الدالية الساكنة Static member functions**:-

يمكننا الوصول إلى البيانات الساكنة من أي عضو دالي في الفئة ولكن من الطبيعي استعمال نوع خاص من الدالات ينطبق على الفئة بأكملها وليس على كائن ما وهو الدالات الساكنة . يتم تعريف العضو الدالي الساكن بواسطة الكلمة الأساسية static لكن استدعاءات هذه الدالة تتم من دون استعمال كائن معين بل يشار إلى الدالة من خلال ربط اسمها باسم الفئة بواسطة عامل دقة المدى:: . لا يستطيع العضو الدالي الساكن الإشارة إلى أي عضو دالي غير ساكن لأن الدالات لا تعرف أي شئ عن كائنات الفئة وكل ما تستطيع الوصول إليه هو بيانات ساكنة ترتبط بالفئة ككل ، لذا يمكننا استدعاء الدالة الساكنة حتى قبل إنشاء أي كائن . ولكن عند استعمال الدوال الساكنة يجب وضع القيود التالية في الاعتبار:-

1/ لا تمتلك الأعضاء الدالية الساكنة المؤشر this .

2/لا يمكن أن يكون هنالك إصدارين من نفس الدالة أحدهما ساكن والآخر غير ساكن .

3/ العضو الدالي الساكن كما سنرى فيما بعد لا يمكن أن يكون افتراضيا virtual.

4/ لا يمكن الإعلان عن الدالة الساكنة على أنها const.

ففي البرنامج تم تعريف الدالة get-resource على أنها ساكنة. يمكن استدعاء الدالة get-resource بذكر اسمها فقط دون أي كائن.

```
//Program 7-9:
```

```
#include<iostream>
class cl {
static int resource;
public:
static int get_resource( );
void free_resource( ) {resource = 0;}
};
int cl :: resource;  //define resource
int cl:: get_resource( )
{
if(resource) return 0 ; // resource alreay in use
else {
  resource = 1;
  return 1; //resource allocated to this object
//Continued
}
}
int main( )
{
cl ob1, ob2;
/* get_resource( ) is static so may be called independent
of any object.*/
if( c1 :: get_resource( )) cout << "ob1 has resource\n ";
if( ! c1 :: get_resource( )) cout << "ob2 denied resource\n
";
ob1.free_resource( );
if(ob2.get_resource(  )) // can still call using object
syntax
cout<<" ob2 can now use resource\n ";
return 0;
}
```

الملخص:

- المشيد هو عضو دالي خاص يحمل اسم الفئة يستعمل لتمهيد الكائنات عند قيم معينة عند إنشاؤها .
- لا يحمل المشيد أي قيمة إعادة.
- المهدم هو عضو دالي يعمل على إلقاء تخصيص الذاكرة التي خصصها المشيد للكائن.
- يحمل المهدم نفس اسم الفئة لكن تسبقه العلامة ~ .
- لا يحمل المهدم أي قيمة إعادة.
- من الممكن تمرير وسائط إلى المشيدات ويتم ذلك بنفس الطريقة التي تمرر بها إلى الدوال الأخرى.
- يتم استدعاء المشيدات كلما تم إنشاء كائن، ويتم استدعاء المهدم لكل كائن قبل تدميره.
- العضو البياني الساكن هو متغير يكون منطبقاً لكل كائنات الفئة.
- تم تمهيد المتغيرات الساكنة عند 0.
- يتم تصريح المتغير الساكن داخل الفئة باستعمال الكلمة الأساسية **static** ويتم تعريفه خارجها.
- يمكن أيضاً تعريف أعضاء دالية ساكنة.
- يتم تعريف العضو الدالي الساكن باستعمال الكلمة الأساسية **static**.
- استدعاءات الأعضاء الدالية الساكنة تتم من دون استعمال كائن معين.
- يشار للدالة من خلال ربط اسمها باسم الفئة من عبر عامل دقة المدى :: .
- لا يستطيع العضو الدالي الساكن الإشارة إلى أي عضو دالي غير ساكن.يمكن جعل كائن تابع لفئة ما ثابتاً إذا كنا نريد ضمان عدم تغير الأعضاء البيانية للكائن.
- للإعلان عن الكائنات الثابتة نستخدم الكلمة الأساسية **const**.
- يمكن تعريف أعضاء دالية ساكنة لا تغير الكائن الذي أستدعي من أجلها.
- لجعل عضو دالي ثابتاً تتم كتابة الكلمة الأساسية **const** في تعريف العضو الدالي وتصريحه مباشرة بعد الأقواس التي نلي اسمه.



الأسئلة

/ **ما هو الخطأ في الجزء التالي من برنامج افترض التصريح الآتي في فئة تدعى Time**:

```
void  ~Time (int) ;
```

7/ ناقش مفهوم الصداقة Friend ship في C++ مع بيان الأوجه السالبة فيها.

8/ هل يمكن أن يحتوى تعريفاً صحيحاً لفئة تدعى Time على كلا المشيدين أدناه:-

```
Time ( int  h = 0, int m = 0, int s = 0);
Time(  );
```

9/ أوجد الخطأ في تعريف الفئة التالي:

```
class Example {
public:
example ( int y = 10)  { data = y ; }
int get Incrementdata ( ) const {
       return ++ data;  }
static get count ( )
{
cout << " data is " << data << endl;
return count;
}
private:
int data;
static int count;
};
```

10/ ماذا يحدث إذا تم تحديد قيمة إعادة لكل من المشيدات والمهدمات حتى ولو كانت void.

.08

الوحدة الثامنة

Classes(III) - (III) الفئات



الأهداف:

بنهاية هذه الوحدة:

- ستتعرف على الغرض من الدوال الصديقة.
- ستتعرف على الفئات الصديقة.
- ستتعرف على كيفية إعادة تعريف العوامل لتعمل مع الأنواع الجديدة.
- ستتعرف على القيود التي تواجه تحميل العوامل بشكل زائد.

.18 الدوال الصديقة
Friend Functions

يمكن لدالة ليست عضواً في فئة ما الوصول إلى الأعضاء الخاصة بتلك الفئة وذلك بجعل الدالة صديقة friend لدوال تلك الفئة .عادة تفترض أعمال التغليف وإخفاء البيانات قاعدة أنه يجب أن لا تكون الدالات التي ليست عضواً في الفئة قادرة على الوصول إلى بيانات الكائن الخاصة والمحمية ، لكن هنالك حالات يؤدى فيها هذا إلى بعض الصعوبات لذا فالدالات الصديقة هي وسيلة للالتفاف حول هذه القاعدة .لجعل دالة ما صديقة نكتب الإعلان عنها داخل الفئة مسبوقاً بالكلمة الأساسية friend . المثال التالى يبين كيف يكون هذا:

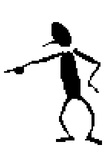
//Program 8-1:

```
#include<iostream.h>
class myclass {
int a, b;
public:
friend int sum(myclass x);
void set_ab(int i,int j);
};
void myclass :: set_ab(int i, int j)
{
a = i;
b =j;
}
// Note: sum( ) is not a member function of any class.
int sum(myclass x)
{
/* Because sum( ) is a friend of myclass, it can directly
access a and b. */
return x.a + x.b;
}
int main( )
{
myclass n;
n.set_ab(3, 4);
cout<<sum(n);
return 0;
```

```
}
```

الخرج من البرنامج:

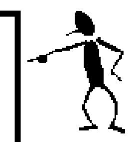
```
7
```



في البرنامج السابق الدالة sum( ) هي ليست عضواً في الفئة myclass ولكن بالرغم من ذلك يمكنها الوصول إلى الأعضاء الخاصة في الفئة my class

ومن الجدير بالذكر أنه على الرغم من أن الدوال الصديقة تزيد من مرونة اللغة C++ إلا أن ذلك لا يتماشى مع فلسفة وجوب السماح للأعضاء الدالية التابعة للفئة فقط الوصول إلى البيانات الخاصة بالفئة ،ومن هنا يبرز السؤال ما هو مدى الخطورة التي تتعرض لها سلامة البيانات عند استعمال دالة صديقة؟

يجب تصريح الدالة على أنها صديقة من داخل الفئة التي ستصل إليها بياناتها ،لذا فالمبرمج الذي لا يستطيع الوصول إلى الشيفرة المصدر للفئة لا يستطيع جعل الدالة صديقة ،وعليه ستبقى سلامة البيانات محافظ عليها وعليه الدالات الصديقة لا تشكل تهديداً خطيراً على سلامة البيانات .



استعمل دائماً عضواً دالياً وليس دالة صديقة إلا إذا كان هنالك سبب قوى يدفع إلى استعمال دالة صديقة كما سنرى لاحقاً.

8-2 الفئات الصديقة Friend Classes

الفئات كما الدالات يمكن أن تكون صديقة والسبب في استعمال دالات صديقة هو تسهيل الاتصال بين الفئات حيث يمكن لفئة صديقة لفئة أخرى الوصول لكل الأعضاء الخاصة المعرفة في الفئة الأخرى . المثال البرنامج يبين هذا:

```
//Program 8-2:
//using a friend class.
```

```
#include<iostream.h>
class TwoValues  {
//continue
int a;
int b;
public:
TwoValues(int i, int j)     {a = i, b= j;}
friend class Min;
};
class Min {
public:
int min(TwoValues x);
};
int Min::min (TwoValues x)
{
return x.a< x.b? x.a: x.b;
}
int main( )
{
TwoValues ob(10, 20);
Min m;
cout<< m.min(ob);
return 0;
}
```

الخرج من البرنامج:

```
10
```

تم الإعلان عن الفئة Min كفئة صديقة للفئة TwoValues في السطرالتالي:

```
friend class Min;
```

لذلك تم الوصول إلى الأعضاء الخاصة a و b في الفئة TowValues من قبل الفئة Min.

```
int Min::min (TwoValues x)
{
return x.a< x.b? x.a: x.b;
}
```

تستعمل الفئات الصديقة إذا كان هنالك فئتين مرتبطتين ببعضهما كثيراً لدرجة أن أحدهما تحتاج إلى الوصول إلى بيانات الأخرى الخاصة بشكل مباشر . أننا لا نريد أن نجعل البيانات عامة لأن هذا سيتيح لأي شخص تعديلها بطريق الخطأ. كما أن الفئة هي ليست مشتركة في صفات مع الفئة الأخرى وعليه لا يمكن استخدام الوراثة لذا فإن استعمال الفئات الصديقة هو الأسلوب الوحيد لجعل إحدى الفئتين تصل إلى الأعضاء الخاصة في الفئة الأخرى.

8.3 تعيين الكائنات Object assignment

يمكن تعيين قيمة كائن إلى كائن آخر باستعمال علامة المساواة = شريطة أن تنتمي هذه الكائنات إلى نفس الفئة ويؤدى هذا إلى أن يحمل الكائن الذي على يسار علامة المساواة قيمة الكائن على يمينها.

البرنامج التالي يوضح ذلك:

```
//Program 8-3:
// Assigning objects.
#include<iostream.h>
class myclass {
int i;
public:
void set_i(int n)   {i=n;  }
int get_i( ) {return i ;}
} ;
int main( )
{
myclass ob1,     ob2;
ob1.set_i(99);
ob2=ob1; // Assign data from ob1 to ob2
cout << " This is ob2's i: " << ob2.get_i( ) ;
return 0;
}
```

الخرج من البرنامج

```
This is ob2's i: 99
```

.48 تحميل العوامل بشكل زائد Operators Overloading

لا تضيف ++C فقط إمكانية استخدام الفئات لإنشاء أنواع جديدة من البيانات بل وتتيح أيضاً للمستخدم العمل على هذه الأنواع باستخدام نفس العوامل التي تستخدمها الأنواع الأساسية . وعندما يعطي عامل موجود أصلاً ك + أو – القدرة على العمل على أنواع بيانات جديدة يقال أنه تم تحميله بشكل زائد overloaded. يتم تحميل العوامل بشكل زائد بكتابة دوال تحمل اسماً خاصاً، الكلمة الأساسية operator متبوعة بالعامل المراد تحميله بشكل زائد ، فمثلاً لتحميل العامل + بشكل زائد نعرف دالة تحمل الاسم ( )+operator.

□ عند تحميل العوامل بشكل زائد يجب مراعاة الآتي:



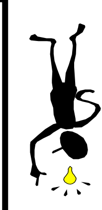
1/ لا يمكن تحميل كل عوامل ++C بشكل زائد ، فمثلاً العوامل التالية لا يمكننا تحميلها :

?: :: .* .

2/ لا يمكن تغيير عدد المعاملات التي يأخذها العامل.

3/ لا يمكن إنشاء عوامل جديدة غير موجودة أصلاً في ++C كالعامل ** الذي يستخدم في بعض اللغات للرفع الأسي.

4/ لا تتغير أولوية precedence العامل المحمل بشكل زائد.



5/ بإمكان العامل المحمل بشكل زائد عند تطبيقه على الكائنات (ليس على الانواع الأساسية) تنفيذ أي عملية يريدها منشئ الفئة ، فمثلاً بإمكان العامل + المحمل بشكل زائد أن يعرض نصاً على الشاشة أو حتى يقوم بطرح كائنين ولكن من المستحسن أن تكون العملية المراد للعامل المحمل بشكل زائد تنفيذها أن تكون لها علاقة بطبيعة العامل أصلاً.

6/ بعض الفئات ملائمة لاستخدام العوامل المحملة بشكل زائد على عكس البعض الآخر ،وبشكل عام يتم استعمال العوامل المحملة بشكل زائد مع الفئات التي تمثل أنواع بيانات رقمية كالأوقات والتواريخ والأرقام المركبة (x+iy) كما يمكن أن تستفيد فئات السلاسل أيضاً من العوامل المحملة بشكل زائد.

.58 كيفية تعريف دالة العامل Operator function

يمكن تعريف الدالة التي تعمل على تحميل عامل بشكل زائد في فئة ما كعضو في الفئة أو كدالة صديقة للفئة.

تأخذ دالة العامل operator function عندما تكون عضواً في الفئة الشكل العام الآتي:

```
return_type operator#(arg_list)
{
 //operations
}
```

حيث :-

return_type : هو قيمة إعادة الدالة operator# والتي غالباً ما ترجع كائناً تابعاً للفئة التي تعمل على كائناتها ولكن يمكن أن يكون return_type من أي نوع آخر.

Operator :- كلمة أساسية في ++C.

:- تستبدل بالعامل المراد تحميله بشكل زائد ، فمثلاً إذا كنا نقوم بتحميل العامل + بشكل زائد نكتب **operator**.

Arg_list:- وهى لائحة الوسيطات الممرة إلى الدالة operator# والتي تحتوى على عنصر واحد إذا كنا نقوم بتحميل عامل ثنائي (+، -، /، ....) وتكون فارغة إذا كنا نقوم بتحميل عامل++ C أحادى (++، -- ، ....).

Operations:- العمليات المراد من العامل المحمل بشكل زائد تنفيذها.

والآن وبعد ان تعرفنا على كيفية كتابة دالة تقوم بتحميل عامل بشكل زائد ، إليك مثالاً مبسطاً يقوم بإنشاء فئة تدعى loc ويقوم بتحميل العامل + ليعمل على كائنات هذه الفئة، أدرس البرنامج وانتبه جيداً إلى كيفية تعريف الدالة .operator+( )

```
//Program 8-4:
#include <iostream.h>
class loc {
int longitude, latitude;
public:
loc() { }
loc(int lg, int lt) {
longitude = lg;
latitude =lt;
}
void show( ) {
```

```
cout << longitude <<" ";
cout<< latitude<< "\n "؛
}
loc operator+ (loc op2)؛
}؛

//Continued
 //Overload +for loc.
Loc loc::operator+(loc op2)
{
loc temp؛
temp.longitude = op2.longitude+ longitude؛
temp.latitude = op2.latitude+ latitude؛
return temp؛
}
int main( )
}
loc ob1(10, 20), ob2(5,30)؛
ob1.show( )؛
ob2.show( )؛
ob1= ob1+ ob2؛
ob1.show( ) ؛
return 0؛
}
```

الخرج من البرنامج:

```
20 10
30 5
15 50
```

لاحظ في الدالة ( )main إن العامل + المحمل بشكل زائد يجعل عملية الجمع تبدو وكأنها تتم على أنواع أساسية .

ob1= ob1+ ob2؛

وكما رأينا في البرنامج الدالة ( )+operator لها وسيطة واحدة على الرغم من أنها تقوم بتحميل عامل الجمع + الثنائي الذي يعمل على قيمتين والسبب في ذلك أن المعامل على يسار العلامة + يتم تمريره إلى الدالة بواسطة المؤشر this والمعامل على يمين العلامة هو الذي يتم تمريره كوسيطة للدالة ولذلك يتم الإعلان عن الدالة كالآتي:

loc operator + (loc op 2)؛

يتم في الدالة ( )main تعيين قيمة الإعادة من الجمع إلى الكائن ob1 ويتم هذا الأمر في الدالة ( )+operator عن طريق إعادة كائن يدعى temp بالقيمة حيث يتم استعمال الكائن temp لتخزين نتائج العمليات الحسابية وهو الكائن الذي تتم إعادته.وبطرق متشابهة يمكننا تحميل العوامل الحسابية الثنائية الأخرى كـ – و * و / بشكل زائد أيضاً .

المثال التالي يقوم بتحميل ثلاث عوامل إضافية في الفئة loc : العامل – والعامل = والعامل ++.

```
//Program 8-5:
#include<iostream.h<
class loc {
int longitude, latitude؛
public:
```

```
loc( )   { }// needed to construct temporaries
loc(int lg, int lt){
longitude = lg;
latitude =lt;
}
void show( )
cout << longitude;"  " >>
cout<< latitude<< "\n";

//Continued
}
loc operator+(loc op2)
loc operator- (loc op2);
loc operator= (loc op2);
loc operator++;
}
 //Overload + for loc.
Loc  loc:: operator+ (loc op2)
{
loc temp;
temp.longitude = op2.longitude+ longitude;
temp.latitude = op2.latitude+ latitude;
return temp;
}
 //Overload - for loc.
Loc  loc:: operator- (loc op2)
{
loc temp;
 //notice order of operands
```

```
temp.longitude = longitude- op2.longitude؛
temp.latitude = latitude- op2.latitude؛
return temp؛
}
 //overload asignment for loc.
Loc  loc:: operator= (loc op2)
{
temp.longitude = op2.longitude؛
//Continued
temp.latitude = op2.latitude؛
return *this;       // i.e., return  object that
                     //generated call
}
 //overload prefix ++ for loc.
Loc  loc:: operator( ) ++
{
longitude++؛
latitude++؛
return *this       ؛
}
int main( )
{
loc ob1(10, 20), ob2(5,30) , ob3(90, 90)؛
ob1.show( )؛
ob2.show( )؛
 ++ob1؛
 ob1.show( ) ؛
ob2 = ++ob1؛
```

```
ob1.show( ) ؛
ob2.show( ) ؛
ob1=ob2=ob3 ؛
ob1.show( )؛
ob2.show( )؛
return 0؛
}
```

الخرج من البرنامج:

```
21 11
22 12
22 13
90 90
90 90
```

في البرنامج السابق:

الدالة ( )-operator:–

```
Loc  loc:: operator- (loc op2)
{
loc temp؛
 //notice order of operands
temp.longitude = longitude- op2.longitude؛
temp.latitude = latitude- op2.latitude؛
return temp؛
}
```

لاحظ في الدالة operator - ( ) ترتيب المعاملات في عملية الطرح.

المعامل على يمين علامة الطرح يتم طرحه من المعامل على يسار علامة الطرح وذلك لأن المعامل على يسار علامة الطرح هو الذي يقوم باستدعاء الدالة operator - ( ) وعليه بيانات الكائن ob2 يتم طرحها من بيانات الكائن المشار إليه بالمؤشر this.

الدالة operator=( ):-

```
Loc  loc:: operator= (loc op2)
{
temp.longitude = op2.longitude؛
temp.latitude = op2.latitude؛
return *this;       // i.e., return  object that
                    //generated call
}
```

في C++ يكون العامل = محملاً بشكل زائد في كل الفئات بشكل افتراضي حتي لو لم تتم كتابة دالة لتحميله . في المثال السابق الدالة operator =( ) تقوم بنفس مهمة العامل = الافتراضي ولكن في بعض الأحيان يمكن للعامل = المحمل بشكل زائد تنفيذ مهام أخرى.

تعيد الدالة this* وهو الكائن الذي استدعى الدالة.

الدالة operator++( ):

```
loc  loc:: operator++( )
{
longitude++؛
latitude++؛
return *this        ؛
{
```

كما لاحظت في البرنامج لا تأخذ الدالة ( )++operator أي وسيطات وذلك لأن العامل ++ أحادى . يتم تمرير المعامل باستعمال المؤشر this.

لاحظ أن كلا الدالتين ( )=operator و ( )++operator تقوم بتغيير قيمة الكائن الذي استدعى الدالة ففي الدالة ( )=operator يتم تعيين قيمة جديدة للكائن على يسار العلامة = والذي قام باستدعاء الدالة وفي الدالة ( )++operator يتم زيادة الكائن الذي استدعى الدالة بمقدار 1 .

.68 تحميل عوامل التعيين بشكل زائد

يمكننا تحميل عوامل التعيين في C++ كـ =- أو =+ تحميلاً زائداً . فمثلاً الدالة التالية تقوم بتحميل العامل =+ تحميلاً زائداً في الفئة loc.

```
loc  loc:: operator+= (loc op2)
{
loc temp؛
longitude = op2.longitude+ longitude؛
latitude = op2.latitude+ latitude؛
return *this؛
}
```

الفرق بين العوامل الثنائية العادية كـ + وبين عوامل التعيين كـ =+ هو أن عوامل التعيين تعدل الكائن الذي تم استدعاؤها من أجله فمثلاً إذا كتبنا:

```
ob1 += ob2؛
```

سيتم استدعاء الدالة ( )+=operator للكائن ob1 ويتم تعديله بجمع ob2 إليه.

.78 تحميل عامل بشكل زائد باستخدام دالة صديقة

يمكننا تحميل عامل بشكل زائد باستخدام دالة صديقة لدوال الفئة المراد تحميل العامل ليعمل على كائناتها وبما أن الدالة الصديقة هي ليست عضواً في الفئة لذا فهي لا تمتلك المؤشر this وعليه يتم تمرير وسيطاتها ظاهرياً ونعني بذلك أن الدالة الصديقة التي تقوم بتحميل عامل ثنائي يتم تمرير وسيطتين لها بينما يتم تمرير وسيطة واحدة للدالة الصديقة التي تقوم بتحميل عامل أحادى .

عندما نقوم بتحميل عامل ثنائي باستخدام دالة صديقة يتم تمرير المعامل على اليسار في الوسيطة الأولى بينما يتم تمرير المعامل على اليمين في وسيطة الدالة الثانية.

المثال التالي يوضح كيفية تعريف دالة صديقة لتحميل العامل +

```
//Program 8-6:
 #include <iostream.h<
class loc{

//Continued
int longitude, latitude؛
public:
loc( )   { }// needed to construct temporaries
loc(int lg, int lt) {
longitude = lg؛
latitude =lt؛
}
void show( ) {
cout << longitude"؛  ">>
cout<< latitude<< "\n" ؛
}
friend loc operator+ (loc op1, loc op2); // now a
friend loc operator- (loc op2)؛
loc operator= (loc op2؛(
```

```
loc operator؛( )++
}؛
 //now , + is overloaded using friend function.
loc operator+ (loc op1, loc op2)؛
{
loc temp؛
temp.longitude =op1.longitude+ op2.longitude؛
temp.latitude = op1.latitude+ op2.latitude؛
return temp؛
}
 //overload - for loc.
Loc  loc:: operator - (loc op2)
{
loc temp؛
 //notice order of operands
temp.longitude = longitude - op2.longitude؛
temp.latitude = latitude- op2.latitude؛
return temp؛
}
 //overload assignment for loc.
Loc  loc:: operator = (loc op2)
{
longitude = op2.longitude؛
latitude = op2.latitude؛
return *this;         // i.e., return  object that generated
call
}
 //overload ++ for loc.
```

```
Loc  loc:: operator++( )
{
longitude؛ ++
latitude؛ ++
return *this        ؛
}
int main( )
{
loc ob1(10, 20), ob2(5,30؛(
ob1 = ob1+ ob2؛
ob1.show ؛( )
return 0؛
}
```

الخرج من البرنامج:

```
50    15
```

- هنالك بعض عوامل C++ لا يمكن تحميلها باستخدام دالة صديقة وهي : = ، ( ) ، [ ] ، > – .

* يضيف استعمال الدوال الصديقة مرونة إلى تحميل العوامل بشكل زائد وذلك للآتي:

أفرض أننا قمنا بتحميل العامل + لجمع كائنات فئة العبارة التالية لا تعمل:

ob1=3+ ob2؛

وذلك لأنه وكما ذكرنا سابقاً الدالة ( )+operator يتم استدعاؤها من قبل الكائن الموجود على يسار العلامة + وتأخذ الكائن على يمين + كوسيطة لها ، وبما أن هـ يجب استدعاء الدوال من قبل الكائنات و 3 ليست عضواً في الفئة لذلك لا يمكننا كتابة عبارة كالعبارة السابقة.

لذلك وعلى الرغم من أنه يمكن جمع عدد صحيح إلى كائن تابع لفئة لا يمكننا جمع كائن إلى رقم صحيح إلا إذا استخدمنا دالة صديقة.

المثال التالي يوضح هذا حيث نقوم في المثال بتعريف إصدارين لدالة صديقة وبالتالي يمكن للكائن أن يظهر على يمين أو يسار العامل.

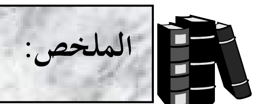
```
//Program 8-7:
 #include <iostream.h>
class loc {
int longitude, latitude؛
public:
loc( ){ }
loc(int lg, int lt) {
longitude = lg؛
latitude =lt؛
}
void show( ) {
cout << longitude<<" " ؛
cout<< latitude<< "\n؛ "
}
friend loc operator+ (loc op1, loc op2)؛
friend loc operator+ (int op1, loc op2) ؛
}
```

```
+ //is overloaded for loc + int.
loc operator+ (loc op1, loc op2);
{
loc temp;
temp.longitude =op1.longitude+ op2;
temp.latitude = op1.latitude+ op2;
return temp;
}
+ //is overload  for int + loc.
loc operator+ (int op1, loc op2);
{
loc temp;
temp.longitude =op1 + op2.longitude;
temp.latitude = op1 + op2.latitude;
return temp;
{
int main( )
{
loc ob1(10, 20), ob2(5,30) , ob3(7, 14);
ob1.show( ) ;
ob2.show( );
ob3.show( );
ob1= ob2 +10;  //both of these
ob3=10 + ob2;  // are valid
ob1.show( );
ob3.show( );
```

```
return 0؛
}
```

الخرج من البرنامج:

```
20 10
30 5
14      7
40     15
40     15
```



الملخص:

- الدوال الصديقة هي دالة ليست عضواً في الفئة ولكنها تستطيع الوصول إلى الأعضاء الخاصة بتلك الفئة.
- لجعل دالة ما صديقة نكتب الإعلان عنها مسبوقاً بالكلمة الأساسية friend .
- يمكن جعل الفئة صديقة لفئة أخرى وذلك لتسهيل الاتصال بين الفئات.
- يمكن تعيين قيمة كائن إلى كائن آخر باستعمال علامة المساواة، شريطة أن تنتمي هذه الكائنات لنفس الفئة.
- عندما يعطى عامل موجود أصلاً القدرة على العمل على أنواع بيانات جديدة يقال أنه تم تحميله بشكل زائد.
- يتم تحميل العوامل بشكل زائد بكتابة دوال تحمل الاسم operator متبوعة بالعامل المراد تحميله بشكل زائد، فمثلاً لتحميل العامل + بشكل زائد نعرف دالة تحمل الاسم operator+( ) .
- يمكن تعريف الدالة التي تعمل على تحميل عامل بشكل زائد في فئة ما كعضو في الفئة أو كدالة صديقة للفئة.
- تأخذ دالة العامل operator function عندما تكون عضواً في الفئة الشكل العام التالي:

```
return_type operator#(arg_list)
{
 //operations
}
```

حيث :-

return_type : هو قيمة إعادة الدالة operator# والتي غالباً ما ترجع كائناً تابعاً للفئة التي تعمل على كائناتها ولكن يمكن أن يكون return_type من أي نوع آخر.

Operator :- كلمة أساسية في C++.

:- تستبدل بالعامل المراد تحميله بشكل زائد ، فمثلاً إذا كنا نقوم بتحميل العامل + بشكل زائد نكتب operator.

Arg_list:- وهى لائحة الوسيطات الممرة إلى الدالة operator# والتي تحتوى على عنصر واحد إذا كنا نقوم بتحميل عامل ثنائي (+، -، /، ....) وتكون فارغة إذا كنا نقوم بتحميل عامل++ C أحادى (++، -- ، ....).

Operations:- العمليات المراد من العامل المحمل بشكل زائد تنفيذها.



الأسئلة

١ – **ناقش مفهوم الصداقة Friend ship في C++ مع بيان الأوجه السالبة فيها.**

٢ – حمل العوامل --& ++ في الفئة stack والتي رأيناها في الأمثلة السابقة بحيث تعمل الدالتان **operator** - -( ) و operator ++ ( ) تماماً مثلما تعمل الدالتان pop( ) و push على التوالي؟

٣ – قم بتحميل العوامل + ، - ، * و / بحيث تقوم بإجراء العمليات الحسابية في فئة تدعى complex تمثل الأعداد المركبة (complex number) التي على الصورة
real part + imaginary part *I

حيث $i = \sqrt{1}$

9.0 الوحدة التاسعة

الوراثة وتعدد الأشكال Inheritance & Polymorphism



بنهاية هذه الوحدة:

ستتعرف على مفهوم الوراثة في لغة C++.

- ستتعرف على كيفية توفير الوراثة لقابلية إعادة استعمال الفئات.
- ستتعرف على مفهوم الفئة القاعدة **(base class)** والفئة المشتقة **(derived class)**.
- ستتمكن من استعمال الوراثة المتعددة لاشتقاق فئة من فئتين قاعدتين أو أكثر.
- ستتعرف على مفهوم تعدد الأشكال **(polymorphism)** في لغة C++ .
- ستتعرف على كيفية الإعلان عن استعمال الدوال الافتراضية **(virtual functions)** .
- ستتعرف على كيفية الإعلان عن استعمال الدوال الافتراضية النقية **(pure virtual functions)** لإنشاء فئات تجريدية **(abstract classes)**.

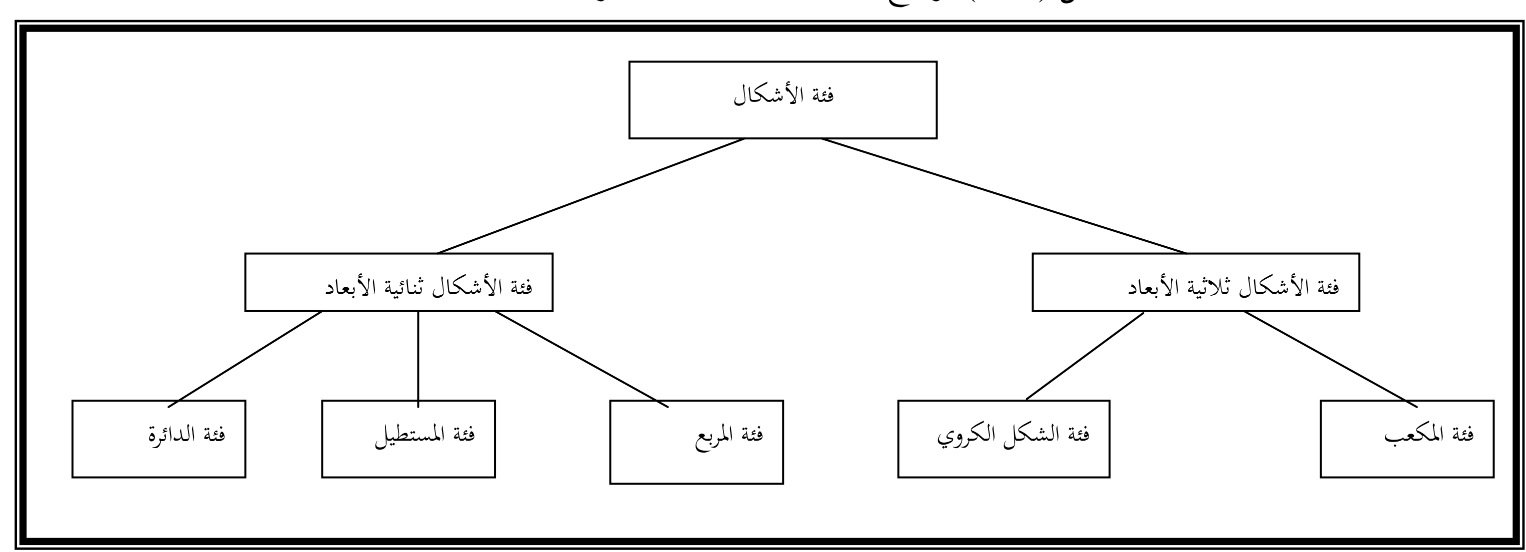
19. مقدمة

الوراثة هي المفهوم الرئيسي بعد الفئات في OOP إلا أنها عملياً تشكل القوة الدافعة لمبدأ البرمجة كائنية المنحى وتعتمد فكرة الوراثة على إمكانية إنشاء فئات جديدة تكون مشتركة في صفات مع فئات موجودة أصلاً وذلك بجعل الفئة الجديدة ترث كل صفات الفئة القديمة بالإضافة إلى صفاتها الخاصة بها فبدلاً من كتابة البيانات والأعضاء الدالية المشتركة مرة أخرى في الفئة الجديدة ترث الفئة الجديدة والتي تسمى بالفئة المشتقة derived class كل البيانات والأعضاء الدالية من الفئة المعرفة أصلاً والتي يرمز لها بالفئة القاعدة **base class**.

عادة تضيف الفئة المشتقة بيانات وأعضاء دالية خاصة بها وعليه تكون الفئة المشتقة أكبر من الفئة القاعدة.

نجد أن كل كائن تابع للفئة المشتقة هو بالضرورة تابع للفئة القاعدة ولكن العكس غير صحيح فكائنات الفئة المشتقة تحمل صفات أكثر من كائنات الفئة القاعدة ، فمثلاً فئة المستطيل هي فئة مشتقة من فئة الأشكال الرباعية وعليه يمكن القول أن أي مستطيل هو شكل رباعي ولا يمكننا القول أن أي شكل رباعي هو مستطيل.

الشكل (1-8) يوضح العلاقة بين الفئة القاعدة والفئات المشتقة.



شكل (1-8) يوضح العلاقة بين الفئة القاعدة والفئات المشتقة

الشكل العام لاشتقاق فئة من فئة قاعدة هو:

```
class derived-class-name : access base-class-name
{
body of class
};
```

تحدد **access** و تسمى محدد وصول إمكانية الوصول إلى أعضاء الفئة القاعدة ،وهى يمكن أن تكون إما **public** أو **private** أو **protected** وإذا لم يتم تحديدها فسيفترض المصرف أن محدد الوصول هو **private** .

عندما يستخدم محدد الوصول **public** تسمى الوراثة عامة، عندما يستخدم المحدد **private** تسمى الوراثة خاصة وعندما يستخدم محدد الوصول **protected** تسمى الوراثة محمية.

إذا كان محدد الوصول عام **public** تسمى الوراثة وراثة عامة وفيها تتم وراثة الأعضاء العامة والمحمية في الفئة القاعدة كأعضاء عامة ومحمية في الفئة المشتقة ولكن في كل الأحوال الأعضاء الخاصة في الفئة القاعدة تبقى خاصة بالفئة القاعدة ولا يمكن الوصول إليها من أعضاء الفئة المشتقة. في البرنامج التالي يتضح لنا أن الكائنات التابعة للفئة المشتقة يمكنها الوصول إلى الأعضاء العامة في الفئة القاعدة إذا كانت الوراثة عامة. لنتابع هذا البرنامج جيداً.

```
//Program 9-1:
#include <iostream.h>
class base {
int i ,  j;
public:
void set( int a , int b) { i= a; j= b;  }
void show( ) { cout<<i <<  "  " << j << "\n"; }
};
class derived : public base {
int k;
public:
derived (int x)  { k=x;  }
void showk( ) { cout << k << "\n" ;   }
};
int main( )
{
derived ob(3);
ob.set(1 ,2); // access member of base
ob.show( ); // access member of base

ob.showk( ); //uses member of derived class
return 0;
}
```

الخرج من البرنامج :

```
1   2
3
```

في البرنامج السابق على الرغم من أن ob هو كائن تابع للفئة derived إلا أنه استطاع الوصول إلى الأعضاء الدالية العامة ( )set و ( )show في الفئة base وذلك لأن الوراثة عامة.

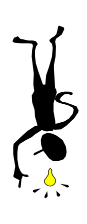
إذا كان محدد الوصول خاص private تسمى الوراثة خاصة وعليه كل الأعضاء العامة والمحمية في الفئة القاعدة تصبح أعضاء خاصة في الفئة المشتقة .

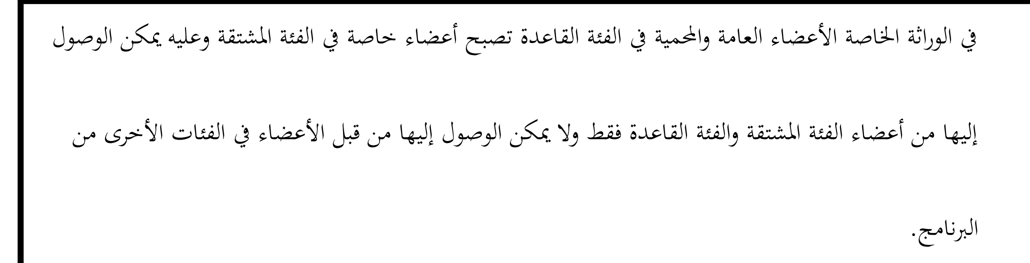
البرنامج التالي لن يعمل وذلك لأن كل من الدوال ( )set و( )show هي الآن خاصة بالفئة القاعدة.

```
//Program 9-2:
// This program won't compile.
#include<iostream.h>
class base {
//Continued
int i ,  j;
public:
void set( int a , int b) { i= a; j= b;  }
void show( ) { cout<<i <<  "  " << j << " \n "; }
};
// Public elements of base are private in derived.
Class derived : private base {
Int k;
Public:
derived (int x)  { k=x;  }
void showk( ) { cout << k << " \n " ;   }
};
int main( )
{
derived ob(3);
ob.set(1 ,2); // error, can't access set( )
ob.show( ); // error, can't access show( )
return 0;
}
```

البرنامج السابق لا يعمل لأن الأعضاء الدالية ( )set و ( )show هي الآن خاصة بالفئة base لأن الوراثة خاصة وبالتالي لا يمكن الوصول إليها من كائن الفئة derived المسمى ob ، وعليه العبارات التالية ليست صحيحة.

```
ob.set(1 ,2);
ob.show( );
```





في الوراثة الخاصة الأعضاء العامة والمحمية في الفئة القاعدة تصبح أعضاء خاصة في الفئة المشتقة وعليه يمكن الوصول إليها من أعضاء الفئة المشتقة والفئة القاعدة فقط ولا يمكن الوصول إليها من قبل الأعضاء في الفئات الأخرى من البرنامج.

الوراثة المحمية

9.2 Protected Inheritance

إذا كان محدد الوصول محمي (protected) تسمى الوراثة محمية وعندها كل الأعضاء العامة والمحمية في الفئة القاعدة تصبح أعضاء محمية في الفئة المشتقة، أي يمكن الوصول إليها من الكائنات في الفئة المشتقة، البرنامج التالي يوضح ذلك:

```
//Program 9-3:
#include <iostream.h>
#include <conio.h>
class base {
protected:
int i ,j ; //private to base , but accessible by derived
public:
void setij( int a , int b) { i= a; j= b;  }
void showij( ) { cout<<i <<  "  " << j << "\n"; }
};
// Inherit base as protected.
class derived : protected base {
int k;
public:
// derived may access base's i and j and setij( ).
void setk( ) { setij( 10, 12) ; k = i*j;  }
//may access showij( ) here
void showall( ) { cout << k<< " "<<endl ; showij( ) ; }
};
int main ( )
{
derived ob ;
// ob.setij(2, 3) ;   // illegal, setij( ) is
//               protected member of derived
ob.setk( ) ; // ok , public member of derived
 ob.showall( ) ;  // ok , public member of derived
//ob.showij( ); // illegal, showij( ) is protected
```

```
                    // member of derived
//Continued
return 0 ;
}
```

الخرج من البرنامج :

```
120
10     12
```

كما رأيت في البرنامج السابق بالرغم من أن الدوال ( )setij و ( )showij هي أعضاء عامة في الفئة base إلا أنها أصبحت محمية في الفئة المشتقة لأننا استخدمنا الوراثة المحمية وعليه لا يمكن الوصول إلى هذه الأعضاء من قبل كائنات الفئة derived.

9.3 الوراثة والأعضاء المحمية
Inheritance and protected members

عندما يتم الإعلان عن عضو في فئة ما على انه محمى protected لا يمكن الوصول إلى هذا العضو من قبل الأعضاء خارج الفئة تماماً كالعضو الخاص private ولكن هنالك استثناء هام ، ففي الوراثة العامة في حين أن العضو الخاص لا يمكن الوصول إليه حتى من الأعضاء في الفئة المشتقة، يمكن الوصول إلى العضو المحمى في الفئة القاعدة من قبل الأعضاء في الفئة المشتقة. وعليه باستخدام محدد الوصول protected يمكنك تعريف أعضاء خاصة بالفئة يمكن الوصول إليها من الكائنات في الفئات المشتقة وإليك البرنامج الذي يوضح ذلك:

```
//Program 9-4:
#include <iostream.h>
class base {
protected:
int i ,j ; //private to base , but accessible by derived
public:
```

```
void set ( int a , int b) { i= a; j= b;  }
//Continued
void show( ) { cout<<i <<  "  " << j << "\n"; }
};
class derived : public base   {
int k;
public:
// derived may access base's i and j
void setk( ) {k=i*j ;}
void showk( ) { cout <<k <<  " \n " ;}
};
int main( )
{
derived ob;
ob.set(2, 3) ;     // ok, known to derived
   ob.show( ) ; // ok, known to derived
ob.setk( );
ob.showk( );
int d;
return 0;
}
```

الخرج من البرنامج:

```
2        3
6
```

في هذا المثال تمت وراثة الفئة derived من الفئة base وراثة عامة و تم الإعلان عن البيانات i و j على أنها محمية العضو الدالي setk( ) في الفئة derived ولذلك يمكن للعضو الدالي الوصول إلى هذه البيانات .

9.4 المشيدات والمهدمات والوراثة

من المهم أن نعرف ترتيب تنفيذ دوال المشيدات والمهدمات عند إنشاء كائن تابع للفئة المشتقة ، لنبدأ بدراسة البرنامج:

```
//Program 9-5:
#include <iostream.h>
class base {
public:
base ( ) { cout << "Constructing base \n";}
~ base( ) { cout << "Destructing base\n" ; }
};
class derived : public base {
public:
derived( ) { cout <<"Constructing derived\n" ; }
~derived( ) { cout<< "Destructing derived\n" ; }
};
int main ( )
{
derived ob;
// do nothing but construct and destruct ob
return 0;
}
```

من التعليق المكتوب في الدالة main( ) يتضح لنا أن البرنامج يشيد ثم يهدم كائناً يدعى ob تابع للمشتقة derived .

فالخرج من البرنامج يكون كالتالي:

```
Constructing base
Constructing derived
Destructing derived
Destructing base
```

كما ترى من خرج البرنامج تم تنفيذ مشيد الفئة القاعدة يليه مشيد الفئة المشتقة ،ولكن تم تنفيذ مهدم الفئة المشتقة قبل مهدم الفئة القاعدة.

وعموماً القاعدة هي:- يتم استدعاء المشيدات بترتيب اشتقاق الفئات (الفئة القاعدة ثم المشتقة ثم المشتقة منها وهكذا) بينما يتم استدعاء المهدمات بعكس ترتيب الاشتقاق ، البرنامج التالي يوضح ذلك:

```
//Program 9-6:
#include<iostream.h>
class base {
public:
base ( ) { cout << " Constructing base \n ";}
~base( ) { cout << " Destructing base\n " ; }
};
class derived1 : public base {
public:
derived1 ( ) { cout " Constructing derived1\n " ; }
~derived1 ( ) { cout " Destructing derived1\n " ; }
};
class derived2 : public derived1 {
public:
derived2 ( ) { cout " Constructing derived2\n " ; }
~derived2 ( ) { cout " Destructing derived2\n " ; }
};
int main ( )
{
derived2 ob;
// construct and destruct ob
return 0;
}
```

الخرج من البرنامج:

```
Constructing base
Constructing derived1
Constructing derived2
Destructing derived2
Destructing derived1
Destructing base
```

9.5 الوراثة المتعددة Multiple Inheritance

تحدث الوراثة المتعددة عندما ترث فئة ما من فئتين قاعدتين أو أكثر كالتالي:

```
class base1
{ };
class base2
{ };
class derived: public base1, public base2
{ };
```

الفئة **derived** مشتقة من الفئتين **base1** و**base2** . يتم في مواصفات الفئة المشتقة فصل الفئات القاعدة عن بعضها البعض بواسطة فاصلة . يجب أن يكون هنالك محدد وصول لكل فئة قاعدة.

البرنامج التالي يبين كيفية استعمال الوراثة المتعددة.

```
//Program 9-7:
// An example of multiple base classes.
#include<iostream.h>
class base1 {
protected:
int x;
```

```
public:
//Continued
void showx( ) { cout << x<< " \n " ; }
};
class base2 {
protected:
int y;
public:
void showy( ) { cout << y<< " \n " ; }
} ;
// Inherit multiple base classes .
class derived: public base1 , public base2 {
public:
void set (int i , int j ) { x=i; y=j ; }
} ;
int main ( )
{
derived ob ;
ob.set(10, 20) ; // provided by derived
ob.showx( ) ;     // from base1
ob.showy( ) ;     //from base2
return 0;
}
```

الخرج من البرنامج:

```
10
20
```

في البرنامج السابق ورثت الفئة derived الفئتين base1 و base2 وراثة عامة، لذلك يمكن للكائن ob الذي يتبع للفئة derived الوصول إلى الأعضاء الدالية العامة showx( ) التابع للفئة base1 و showy( ) التابع للفئة base2 .

9.6 تعدد الأشكال

Polymorphism

هنالك ثلاثة مفاهيم رئيسية في البرمجة الكائنية المنحى . الأول هو الفئات والثاني الوراثة سنناقش هنا المفهوم الثالث : تعدد الأشكال الحقيقي يتم تطبيقه في C++ من خلال الدالات الإفتراضية**virtual functions**.

يوجد في الحياة الفعلية مجموعة من الأنواع المختلفة من الأشياء والتي عند إعطائها تعليمات متطابقة تتصرف بطرق مختلفة ، في C++ عادة يحدث تعدد الأشكال في الفئات المرتبطة ببعضها البعض بسبب الوراثة وهذا يعنى أن استدعاء عضو دالي سيؤدى إلى تنفيذ دالة مختلفة وفقاً لنوع الكائن الذي استدعى العضو الدالي.

يبدو تعدد الأشكال شبيهاً بتحميل الدالات بشكل زائد ، لكن تعدد الأشكال آلية مختلفة وأكثر فعالية فعند تحميل الدالات بشكل زائد المصرف هو الذي يحدد الدالة التي سيتم تنفيذها بينما في تعدد الأشكال يتم اختيار الدالة المطلوب تنفيذها أثناء تشغيل البرنامج.

9.7 الدالات الافتراضية

Virtual Functions

هي دوال يتم تعريفها ضمن الأعضاء الدالية في فئة قاعدة **base** ويعاد تعريفها في الفئات المشتقة. لإنشاء **virtual function** تقوم الفئة المشتقة بإعادة تعريف الدالة بما يتوافق مع متطلباتها .

*** عندما يعلن عن مؤشر ليشير إلى كائنات فئة قاعدة يمكن استخدام نفس المؤشر ليشير إلى كائنات الفئات المشتقة وعليه عندما يشير مؤشر فئة قاعدة إلى كائن في فئة مشتقة منها تحتوى على **virtual function** تحدد C++ الدالة المطلوب تنفيذها وفقاً لمحتويات المؤشر (نوع الكائن المشار إليه بواسطة المؤشر) ويتم هذا التحديد أثناء تنفيذ البرنامج وعليه عندما يستعمل مؤشر الفئة القاعدة ليشير إلى كائنات الفئات المشتقة يتم تنفيذ عدة إصدارات من الدالة الإفتراضية بناءاً على محتويات المؤشر.

البرنامج التالي يوضح ذلك:

Program 9-8:

```
#include<iostream.h>
class base {
//Continued
public:
virtual void vfunc( ) {
cout << " This is base's vfunc( ) .\n ";
}
};
class derived1 : public base {
public :
void vfunc( )  {
cout << " This is derived1's vfunc( ) .\n ";
}
};
class derived2 : public base {
public  :
void vfunc( )  {
cout << " This is derived2's vfunc( ) .\n ";
}
};
int main( )
{
base *p, b;
derived1 d1;
derived2 d2;

// point to base
p= &b;
p->vfunc( ) ; // access base's vfunc( )
// point to derived1
p= &d1;
p->vfunc( ) ; // access derived1's vfunc( )
// point to derived2
```

```
p= &d2;
p->vfunc( ) ; // access derived2's vfunc( )
return 0;
}
```

الخرج من البرنامج:

```
This is base's vfunc( ).
This is derived's vfunc( ).
This is derived's vfunc( ).
```

داخل الفئة base تم تعريف الدالة الإفتراضية ( )vfunc . لاحظ أن الكلمة الأساسية virtual تسبق اسم الدالة في الإعلان عنها . تم إعادة تعريف الدالة ( )vfunc في الفئات المشتقة derived1 و derived2 .

داخل الدالة main تم الإعلان عن أربعة متغيرات:-

اسم المتغير	نوعه
p	مؤشر لكائنات الفئة القاعدة base
b	كائن تابع للفئة base
d1	كائن تابع للفئة derived1
d2	كائن تابع للفئة derived2

تم تعيين عنوان الكائن b إلى المؤشر p وتم استدعاء الدالة ( )vfunc بواسطة المؤشر p وبما أن المؤشر الآن يحمل عنوان الكائن التابع للفئة base تم تنفيذ إصدار الدالة vfunc( ) المعرف في الفئة base .بعدها تم تغيير قيمة المؤشر p إلى عنوان الكائن d1 التابع للفئة المشتقة derived1 الآن سيتم تنفيذ الدالة

derived1:: vfunc( )

أخيراً تم تعيين عنوان الكائن d2 التابع للفئة derived2 إلى المؤشر p وعليه العبارة:

p -> func( );

أدت إلى تنفيذ الدالة

derived2:: vfunc( )

من النظرة الأولى قد تبدو الدوال الإفتراضية شبيهة بتحميل الدالات بشكل زائد .
ولكن عند تحميل الدالات بشكل زائد يجب أن يختلف الإعلان عن الدالة من دالة إلى أخرى في نوع أو عدد الوسائط الممررة إلى الدالة حتى يستطيع المصرف تحديد الدالة المطلوب تنفيذها ، بينما في الدوال الإفتراضية يجب أن يطابق إعلان الدالة الإفتراضية المعرفة في الفئة القاعدة الإعلان عنها في الفئات المشتقة.

تذكر دائماً أن الدالة الإفتراضية:

- 1/ لا يمكن أن تكون عضواً ساكناً في الفئة static member.
- 2/لا يمكن أن تعرف كدالة صديقة friend function.
- 3/ لا يمكن استعمالها كمشيد constructor.

9.8 الفئات التجريدية Abstract Classes

تشكل الفئات التجريدية مفهوماً قوياً في *OOP* . الفئة التي لا يتم إنشاء أي كائنات منها تسمى فئة تجريدية . الهدف الوحيد لهذه الفئة هو أن تلعب دور فئة عامة يتم اشتقاق فئات أخرى منها.

9.9 الدالات الإفتراضية النقية Pure virtual functions

سيكون من الجيد لو استطعنا في حال إنشاء فئة قاعدة تجريدية أن نبلغ المصرف أن يمنع أي مستخدم للفئة من إنشاء كائن تابع لها ، يتم ذلك من خلال تعريف دالة إفتراضية نقية واحدة على الأقل في الفئة.

الدالة الإفتراضية النقية هي دالة ليس لها جسم ، يتم إزالة جسم الدالة الإفتراضية في الفئة القاعدة.

الصورة العامة لها:

```
virtual type functionname (parameter-list) = 0;
```

علامة المساواة ليس لها أي علاقة بالتعيين فالتركيب المنطقي **(=0)** هو فقط إبلاغ المصرف أن الدالة ستكون نقية أي لن يكون لها جسم.

البرنامج التالي يحتوى على مثال بسيط لدالة إفتراضية نقية. الفئة القاعدة **number** هي فئة تجريدية تحتوى على عضو محمى من النوع **int** يدعى **val** ، الدالة **setval ( )** ، الدالة النقية **show( )** . في الفئات المشتقة **hextype** ، **oct type** تم إعادة تعريف الدالة **show( )**.

```
//Program 9-9:
#include <iostream.h>
//Continued
class number {
protected :
int val ;
//Continued
public :
void setval (int i) { val = i ; }
// show( ) is a pure virtual function
virtual void show( ) = 0 ;
};
class hextype : public number {
public :
void show ( )    {
cout << hex << val << "\n " ;
}
};
class dectype : public number {
public :
void show ( )    {
cout <<  val << "\n " ;
}
};
class octtype : public number {
public :
```

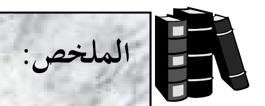
```
void show ( )    {
cout <<  oct << val << "\n " ;
}
};
int main ( )
{
dectype d;
hextype h;
octtype 0;

d.setval(20) ;
d.show( ) ;
h.setval(20) ;
h.show( ) ;
0.setval(20) ;
0.show( ) ;

return 0;
}
```

الخرج من البرنامج:

```
20
14
24
```



الملخص:

- الشكل العام لاشتقاق فئة من فئة قاعدة هو:

```
class derived-class-name : access base-class-name {
body of class
};
```

- تسمى **access** محدد وصول ، وهي تتحكم في كيفية طريقة وراثة الفئات حيث يمكن أن تكون الوراثة عامة (public) أو خاصة (private) أو محمية (protected) على حسب محدد الوصول المستخدم.
- إذا كان محدد الوصول عام تسمى الوراثة عامة وفيها تتم وراثة الأعضاء العامة والمحمية في الفئة القاعدة كأعضاء عامة ومحمية في الفئة المشتقة ولكن تبقى الأعضاء الخاصة في الفئة القاعدة خاصة بالفئة القاعدة، ولا يمكن الوصول إليها من أعضاء الفئة المشتقة.
- إذا كان محدد الوصول خاص تسمى الوراثة خاصة وعندها كل الأعضاء العامة والمحمية في الفئة القاعدة تصبح أعضاء خاصة في الفئة المشتقة.
- إذا كان محدد الوصول محمى تسمى الوراثة محمية وعندها كل الأعضاء العامة والمحمية في الفئة القاعدة تصبح أعضاء محمية في الفئة المشتقة.
- لا يمكن الوصول إلى العضو المحمى من قبل الأعضاء خارج الفئة إلا أنه في الوراثة العامة يمكن الوصول إلى العضو المحمى من الأعضاء في الفئات المشتقة.
- عادة يتم تنفيذ مشيد الفئة القاعدة ثم مشيد الفئة المشتقة ولكن يتم تنفيذ مهدم الفئة المشتقة أولاً قبل مهدم الفئة القاعدة.
- تحدث الوراثة المتعددة عندما ترث فئة ما من فئتين قاعدتين أو أكثر.
- يحدث تعدد الأشكال عادة في الفئات المرتبطة ببعضها بسبب الوراثة.
- الدوال الافتراضية هي دوال يتم تعريفها ضمن الأعضاء الدالية في الفئة القاعدة ويعاد تعريفها في الفئات المشتقة.
- عندما يشير مؤشر فئة قاعدة إلى كائن في فئة مشتقة منها تحتوى على دالة افتراضية، تحدد ***C++*** الدالة المطلوب تنفيذها وفقاً لمحتويات المؤشر ويتم ذلك أثناء تنفيذ البرنامج.
- يجب أن نطابق إعلان الدالة الافتراضية في الفئة القاعدة بالإعلان عنها في الفئات المشتقة.
- الفئة التجريدية (abstract class) هي الفئة التي لا يتم إنشاء أي كائنات منها.

♦ الدالة الافتراضية النقية هي دالة ليس لها جسم يتم تعريفها في الفئات التجريدية.

الأسئلة

1/ أكتب تعريفاً مختصراً لكل من الآتي:

- الوراثة **(Inheritance)**.
- الوراثة المتعددة **(multiple inheritance)**.
- الفئة القاعدة **(base class)**.
- الفئة المشتقة **(derived class)**.

2/ (صحيح / خطأ) : كائن الفئة المشتقة هو أيضاً كائن تابع للفئة القاعدة لها.

3/ يفضل بعض المبرمجين عدم استعمال محدد الوصول المحمى (protected) لأنه يهدد سلامة بيانات الفئة القاعدة . ناقش هذه العبارة وبين ما مدى صحتها .

4/ ما هي الدوال الافتراضية ؟ صف الأحوال التي تكون فيها استعمال الدوال الافتراضية مناسباً؟

5/ وضح الفرق بين الدوال الافتراضية والدوال الافتراضية النقية (pure) .

6/ (صحيح / خطأ) كل الدوال الإفتراضية في الفئات القاعدة التجريدية (abstract base classes) يجب أن تكون دوال افتراضية نقية.

9.0

الوحدة العاشرة

القوالب والإستثناءات



- ستتمكن من استعمال قوالب دالات لإنشاء مجموعة من الدوال المرتبطة ببعضها.
- ستتمكن من استعمال قوالب الفئات **(Templates Classes)**.
- ستتعرف على مفهوم الاستثناءات في لغة *C++*.
- ستتمكن من استعمال كتل المحاولة **try blocks** والتي تحصر العبارات التي يمكن أن تؤدى إلى حدوث استثناء.
- ستتمكن من رمى الاستثناء.
- ستتمكن من استعمال كتل التقاط **catch blocks** والتي تقوم بمعالجة الاستثناء.

.19 قوالب الدالات
Template Functions

إذا أردنا كتابة دالة تقوم باستبدال رقمين تتم كتابة هذه الدالة لنوع بيانات معين كالآتي:

```
int swap (int &a,int &b)
{
int temp;
temp=a;
a=b;
b=temp;
}
```

يتم تعريف الدالة من النوع **int** وتعيد قيمة من نفس النوع . لكن لنفترض أننا نريد استبدال رقمين من النوع **long** سنضطر لكتابة دالة جديدة كلياً.

```
Long swap (long &a, long &b)
{
long temp;
temp=a;
a=b;
b=temp;
}
```

وسنضطر لكتابة دالة أخرى إذا أردنا استبدال رقمين من النوع **float** .

إن جسم الدالة هو نفسه في كل الحالات لكن يجب أن تكون دالات منفصلة لأننا نتعامل مع متغيرات ذات أنواع مختلفة وعلى الرغم من أنه يمكن تحميل هذه الدالات بشكل زائد بحيث تحمل نفس الاسم لكننا أيضاً نضطر إلى كتابة دالات منفصلة لكل نوع وهذه الطريقة بها عدة عيوب :-

1/ كتابة نفس جسم الدالة مراراً وتكراراً لأنواع مختلفة من البيانات يضيع الوقت ويزيد حجم البرنامج .

2/ إذا ارتكبنا أي خطأ في إحدى هذه الدالات يجب تصحيح هذا الخطأ في بقية الدالات.

كانت هنالك طريقة لكتابة هذه الدالة مرة واحدة فقط لكي تعمل على أي نوع من أنواع البيانات المختلفة ويتم هذا باستعمال ما يسمى بقالب الدالات **Functions Templates** والذي يتم إنشاؤها باستخدام الكلمة الأساسية **template** .

البرنامج التالي يبين كيفية كتابة دالة تقوم باستبدال قيمتي متغيرين كقالب لكي تعمل مع أي نوع أساسي .يعرف البرنامج إصدار قالب الدالة **swapargs( )** ثم يستدعى هذه الدالة في**main( )** ثلاث مرات مع أنواع بيانات مختلفة.

```
//Program 9-1:
// Function template example.
// Function template example.
#include <iostream.h>
// This is a function template.
template <class x> void swapargs(x &a, x &b)
{
x temp;
temp = a;
a = b;
b = temp;
}
int main( )
{
int i=10 , j=20;
double x=10.1, y=23.3;
char a= 'x' ,b= 'z' ;

cout << " original i, j: ";
cout<<i<<" "<<j<< "\n " ;
cout << " original x, y:" <<x<<" "<<y<< "\n " ;
cout << " original a, b: " << a <<" "<< b << "\n " ;

swapargs(i, j) ; // swap integers
swapargs(x, y) ; // swap floats
swapargs(a, b) ; // swap chars
cout << " Swapped i, j: "<<i<<" "<<j<< "\n " ;
cout << " Swapped x, y: "<<x<<" "<<y<< "\n " ;
```

```
cout << " Swapped a, b: " <<a<<" "<<b<< "\n " ;
return 0;
}
```

الخرج من البرنامج:

```
original i, j: 10   20
original x, y:  10.1    23.3
original a, b:   x    z
Swapped i, j:   20       10
Swapped x, y:  23.3    10.1
Swapped a, b:   z       x
```

كما رأيت في البرنامج أعلاه تعمل الدالة ( )swapargs الآن مع كل أنواع البيانات int، double، char واستخدام استعملتها كوسائط لها ويمكن أن تعمل أيضاً مع أنواع أساسية أخرى وحتى مع أنواع البيانات المعرفة من قبل المستخدم ، ولجعل الدالة تقوم بكل هذا كتبنا:

```
template< class x> void swapargs (x& a, x&b)
{
x temp;
temp = a;
a = b;
b = temp;
}
```

الابتكار في قوالب الدالات هو عدم تمثيل نوع البيانات الذي تستعمله الدالة كنوع معين int مثلاً ، بل باسم يمكنه أن يشير إلى أي نوع من قالب الدالات في المثال السابق ، هذا الاسم هو x وهو يسمى وسيطة قالب.

9.2 المصرف وقوالب الدالات

عندما يرى المصرف الكلمة الأساسية **template** وتعريف الدالة الذي يليها لا يقوم بتوليد أي شفرة لأنه لا يعرف بعد ما هو نوع البيانات الذي سيستعمل مع الدالة . يتم توليد الشفرة بعد استدعاء الدالة في عبارة ما في البرنامج ، يحصل هذا الأمر في البرنامج السابق في العبارة ;swapargs(i,j) مثلاً.

عندما يرى المصرف مثل هذا الاستدعاء، فانه يعرف أن النوع الذي سيتم استعماله هو **int** كوننا عرفنا المتغيرات **i** و **j** على أنها من النوع **int**.لذا يقوم بتوليد إصداراً للدالة ( )swapargs خاصاً بالنوع **int** مستبدلاً الاسم **x** في كل ظهور له في القالب بالنوع **int** ويسمى هذا استنباط (instantiating) قالب الدالات. كل إصدار مستنبط للدالة يسمى دالة قوالبية.

بشكل مماثل يؤدى الاستدعاء (swapargs(x,y إلى جعل المصرف يولد إصداراً للدالة ( )swapargs يعمل على النوع **double** بينما يؤدى الاستدعاء (swapargs(a,b إلى توليد دالة تعمل على النوع **char**.

يقرر المصرف كيفية تصريف الدالة على أساس نوع البيانات المستعمل في وسيطات استدعاء الدالة . مما سبق يتضح لنا أن قالب الدالات هو ليس في الواقع دالة، إنه مخطط لإنشاء عدة دالات ويتلائم هذا مع فلسفة *OOP* وهو متشابه للفئة كونها نموذج لإنشاء عدة كائنات متشابهة.

9.3 قالب دالات مع وسيطتي قالب

يمكن تعريف أكثر من وسيطة قالب في قالب الدالات وذلك باستعمال فاصلة (،) تفصل بين الوسائط. البرنامج التالي يقوم بإنشاء قالب دالات له وسيطتين

```
//Program 9-2:
#include <iostream.h>
template <class type1,class type2>
void myfunc(type1 x, type2 y)
{
cout <<x<< y << '\n' ;
}
int main( )
```

```
{
myfunc ( 10, " I like C++");
myfunc(98.6, 19L);
return 0;
}
```

في البرنامج السابق تم استبدال type1 وtype2 بأنواع البيانات int، char*، double، long على التوالي.

الخرج من البرنامج:

```
10  I like C++
98.6  19L
```

9.4 قوالب الفئات Templates Classes

الفئة stack والتي سبق أن رأيناها في الأمثلة السابقة كان بإمكانها تخزين بيانات من نوع أساسي واحد فقط هو النوع int ولذلك إذا أردنا تخزين بيانات من النوع float في فئة stack سنحتاج إلى تعريف فئة جديدة كلياً وبشكل مماثل سنحتاج إلى إنشاء فئة جديدة لكل نوع بيانات نريد تخزينه ، لذا علينا كتابة مواصفات فئة واحدة تعمل مع متغيرات من كل الأنواع وليس مع نوع بيانات واحد، بإمكان قوالب الفئات تحقيق ذلك.

المثال يقوم بتعريف الفئة stack باستعمال قالب دالات:

```
//Program 9-3:
// This function demonstrates a generic stack.
#include <iostream.h>
#include <conio.h>

const int SIZE = 10;
```

```
// Create a generic stack class
template <class StackType> class stack  {
StackType stck[SIZE];   // holds the stack
int tos ; // index of top_of_stack

public:
stack( ) { tos =0;  }  // initialize stack
//Continued
void push(StackType ob) ; // push object on stack
StackType pop( ) ;   // pop object from stack
};

//push an object.
template <class StackType> void stack <StackType> ::
push(StackType ob)
{
if (tos== SIZE) {
cout << "Stack is full.\n" ;
return ;
}
stck[tos] = ob;
tos++;
}
//pop an object.
template <class StackType> StackType stack <StackType>
:: pop( )
{
if (tos== 0) {
cout <<  "Stack is empty.\n" ;
return 0;   //return null on empty stack
}
tos--;
return stck[tos];
}
```

```
int main( )
{
// Demonstrate character stacks.
stack<char> s1, s2;       // create two character stacks
int i;

s1.push( 'a' );
s2.push( 'x' );
//Continued
s1.push( 'b' );
s2.push( 'y' );
s1.push( 'c' );
s2.push( 'z' );
for (i=0; i<3; i++ ) cout<<" "  <<s1.pop( ) ;
cout <<endl;
for (i=0; i<3; i++ ) cout<< " " <<s2.pop( )  ;
cout<<endl;
// demonstrate double stacks
stack<double> ds1, ds2;       // create two double stacks
ds1.push( 1.1 );
ds2.push( 2.2 );
ds1.push( 3.3 );
ds2.push( 4.4 );
ds1.push( 5.5);
ds2.push( 6.6 );
for (i=0; i<3; i++ ) cout <<" "<<ds1.pop( ) ;
cout<<endl;
for (i=0; i<3; i++ ) cout<<" "  <<ds2.pop( ) ;
return 0;
}
```

الخرج من البرنامج:

```
c       b       a
```

z	y	x
5.5	3.3	1.1
6.6	4.4	2.2

تم تمثيل الفئة **stack** هنا كقالب فئات، هذا الأسلوب مشابه للأسلوب المستعمل مع قوالب الدالات . تشير الكلمة الأساسية **template** إلى أن الفئة بأكملها ستكون قالباً ويتم عندها استعمال وسيطة قالب تدعى **StackType** .



تختلف قوالب الفئات عن قوالب الدالات في طريقة استنباطها. لإنشاء دالة فعلية من قالب دالات يتم استدعائها باستعمال وسيطات من نوع معين ،لكن الفئات يتم استنباطها بتعريف كائن باستعمال وسيطة القالب :-

```
stack <char> s1, s2;
```

تنشئ هذه العبارة كائنين **s1**، **s2** تابعين للفئة **stack** ويزود المصرف مساحة من الذاكرة لبيانات هذين الكائنين والتي هي من النوع **char** ليس هذا فقط بل وينشئ أيضاً مجموعة من الأعضاء الدالية التالي تعمل على النوع **char**.

لاحظ هنا أن اسم الكائنين يتكون من اسم الفئة **stack** إضافة إلى وسيطة القالب **<char>** مما يميزها عن كائنات بقية الفئات التي قد يتم استنباطها من نفس القالب ك **stack <double>** مثلاً.

الإستثناءات
9.5 (Exceptions)

تزود الإستثناءات أسلوباً كائني المنحى لمعالجة أخطاء التشغيل التي تولدها فئات C++ ، ولكي تكون إستثناءاً يجب أن تحدث تلك الأخطاء كنتيجة لعمل ما جرى ضمن البرنامج كما يجب أن تكون أخطاء يستطيع البرنامج اكتشافها بنفسه .

التركيب النحوي للاستثناء:-

لنفترض أن برنامجاً ما ينشئ كائنات تابعة لفئة معينة ويتفاعل معها ، لا تسبب استدعاءات الأعضاء الدالية أي مشاكل لكن قد يرتكب البرنامج في بعض الأحيان أخطاء مما يؤدى إلى اكتشاف خطأ في عضو دالي ما.

يقوم العضو الدالي عندها بإبلاغ البرنامج أن خطأ ما قد حصل، يسمى هذا الأمر رمى استثناء ويحتوى البرنامج على جزء منفصل لمعالجة الخطأ، يسمى هذا الجزء معالج الاستثناء أو كتلة الالتقاط لأنها تلتقط الإستثناءات التي ترميها الأعضاء الدالية. وأي عبارات في البرنامج تستعمل كائنات الفئة تكون موجودة داخل كتلة تسمى كتلة المحاولة وعليه الأخطاء المولدة في كتلة المحاولة سيتم التقاطها في كتلة الالتقاط .

يستعمل الاستثناء ثلاث كلمات أساسية جديدة throw، catch، try .

البرنامج يوضح ميزات آلية الاستثناء هذه (هو فقط تخطيط عام لإظهار التركيب المنطقي للاستثناء):-

```
//Program 9-4:
class any class
{
public:
class an error
{
};
void func( )
{
if ( /* Error condition*/)
throw an Error( );
}
```

```
};
void main( )
//Continued
{
try
{
any class obj1;
obj1.func( );
}
catch(any class:: An Error)
{
// tell user about the Error
}
}
```

يبدأ هذا البرنامج بفئة تدعى anyclass وهى تمثل أي فئة يمكن أن تحدث فيها أي أخطاء. يتم تحديد فئة الاستثناء في الجزء العام من الفئة any class. تقوم الأعضاء الدالية التابعة للفئة any class بالتدقيق بحثاً عن أي خطأ . إذا وجد تقوم برمي استثناء باستعمال الكلمة الأساسية throw يليها المشيد التابع لفئة الخطأ throw AnError( ) .

قمنا في main( ) بحصر العبارات التي تتفاعل مع الفئة any class في كتلة محاولة إذا سببت أي واحدة من تلك العبارات اكتشاف خطأ في عضو دالي تابع للفئة any class سيتم رمى استثناء وينتقل التحكم إلي كتلة الالتقاط التي تلي المحاولة مباشرة.

البرنامج التالي يستعمل الاستثناءات :-

```
//Program 9-5:
// Demonstrated Exceptions
#include <iostream.h>
#include <conio.h>
const int SIZE =3;
class stack
{
private:
int tos;
```

```
int stck[SIZE];
public:
class Range {  };
//Continued
stack( ) { tos = 0;  }
~stack( ){ };
void push (int i);
int pop( );
};
 void stack::push(int i)
{
if( tos >= SIZE)
throw Range ( );
else
{ stck[tos] = i;
tos ++;
} }
stack :: pop( )
{ if( tos == 0)
  throw Range( );
  else  {
  tos --;
return stck[tos];
} }
main ( )
{ stack s1;
try
{ s1.push(1);
  s1.push(2);
//Continued
s1.push(3);
cout << s1.pop ( )<< endl;
cout << s1.pop ( )<<  endl;
cout << s1.pop ( )<<  endl;
```

```
 cout << s1.pop ( )<< endl;
}
catch (stack::Range)
{
cout << "Stack Full or Empty" << endl;
}
return 0;
 }
```

في البرنامج السابق عبارتين تتسببان في رمي استثناء إذا حذفنا رمز التعليق الذي يسبقهما، اختبر الحالتين. سترى في كلاهما رسالة الخطأ التالية:-

Stack Full or Empty

يحدد البرنامج أولاً جسم فارغ الدالة لأن كل ما نحتاج إليه هو فقط اسم الفئة الذي يتم استعماله لربط عبارة الرمي throw بكتلة الالتقاط.

يحدث الاستثناء في الفئة stack إذا حاول البرنامج سحب قيمة عندما يكون الـ stack فارغاً أو حاول دفع قيمة عندما يكون ممتلئاً .

ولإبلاغ البرنامج أنه قد ارتكب خطأ عند عمله مع كائن stack تدقق الأعضاء الدالية التابعة للفئة stack بحثاً عن خطأ باستعمال عبارات if وترمي استثناءاً إذا حدثت إحدى تلك الحالات . يتم في البرنامج السابق رمي استثناء في مكانين كلاهما باستعمال العبارة:

throw range( );

تقوم range( ) باستحضار المشيد (الضمني) التابع للفئة range الذي ينشئ كائناً تابع لهذه الفئة بينما تقوم throw بنقل تحكم البرنامج إلى معالج الاستثناءات، كل العبارات في main والتي قد تتسبب في هذا الاستثناء محصورة بين أقواس حاصرة وتسبقها الكلمة الأساسية try .

الجزء من البرنامج والذي يعالج الاستثناء موجود بين أقواس حاصرة وتسبقه الكلمة الأساسية catch مع وجود اسم فئة الاستثناء في أقواس .

يجب أن يشتمل اسم فئة الاستثناء على الفئة التي يتواجد فيها.

catch(stack:: range)

يدعى هذا المشيد معالج استثناء ويجب أن يلي كتلة المحاولة مباشرة وهو يقوم في البرنامج السابق بعرض رسالة خطأ فقط لكي يعلم المستخدم عن سبب توقف البرنامج عن العمل .

ينتقل التحكم بعدها إلي ما بعد معالج الاستثناء لكي يستطيع متابعة البرنامج أو يرسل التحكم إلى مكان آخر أو ينهى البرنامج إذا لم تكن هنالك طريقة أخرى .

الخطوات التالية تلخص عملية الاستثناء:-

1/ يتم تنفيذ البرنامج بشكل طبيعي خارج كتلة المحاولة .

2/ ينتقل التحكم إلى كتلة المعالجة.

3/ عبارة ما في كتلة المحاولة تسبب خطأ دالي .

4/ يرمي العضو الدالي استثناء.

5/ ينتقل التحكم إلى كتلة الالتقاط التي تلي كتلة المحاولة.

البرنامج التالي أيضاً يقوم برمي استثناء إذا حاول المستخدم إدخال رقم سالب **negative**.

```
//Program 9-6:
// Catching class type exeptions.
# include <iostream.h>
# include <string.h>
#include <conio.h>
class MyException {
public:
char str_what[80];
int what;
MyException( )    {  *str_what =0 ; what = 0;  }
MyException(char *s, int e )  {
strcpy (str_what, s);
what = e;
}
};
int main( )
{
int i;
try  {
 cout << " Enter a positive number: " ;
cin >> i ;
if (i<0)
throw MyException ("Not Positive" ,i) ;
```

```
}
catch (MyException e)   {  // catch an error
cout <<e.str_what << ":  " ;
cout << e.what << "\n" ;
}
getch();
return 0;
}
```

الخرج من البرنامج بافتراض أن المستخدم قد أدخل i= -4:



```
Enter a positive number:  -4
Not Positive:  -4
```

في البرنامج السابق يطلب البرنامج من المستخدم إدخال رقم موجب، ولكن إذا تم إدخال رقم سالب يقوم البرنامج بإنشاء كائن تابع للفئة My Exception لوصف هذا الخطأ.



الملخص:

- ♦ قوالب الدالات هو وسيلة لجعل الدالة تعمل على أي نوع من أنواع البيانات المختلفة.
- ♦ يتم إنشاء قالب الدالات باستخدام الكلمة الأساسية Template.
- ♦ في قالب الدالات لا يتم تمثيل نوع بيانات معين في الدالة ك **int** مثلاً بل باسم يمكن أن يشير إلى أي نوع بيانات ويسمى هذا الاسم وسيطة قالب.
- ♦ يحدد المصرف كيفية تصريف الدالة على أساس نوع البنيات المستعمل في وسيطات استدعائها.
- ♦ قالب الدالات هو ليس في الواقع دالة، هو مخطط لإنشاء عدة دالات.
- ♦ يمكن تعريف أكثر من وسيطة قالب في قالب الدالات.
- ♦ قالب الفئات هو فئة تعمل على متغيرات في كل أنواع البيانات.
- ♦ تتبع الاستثناءات أسلوباً كائني المنحى لمعالجة أخطاء التشغيل التي تولدها الفئات في *C++*.
- ♦ عند حدوث خطأ في إحدى الفئات تقوم الأعضاء الدالية بإبلاغ البرنامج أن خطأ ما قد حدث ويسمى هذا الأمر رمى استثناء.
- ♦ يحتوى برنامج ***C++*** على جزء منفصل لمعالجة الأخطاء يسمى معالج الاستثناء أو كتلة الالتقاط.
- ♦ أي عبارات في البرنامج تستعمل كائنات الفئة تكون موجودة داخل كتلة تسمى كتلة المحاولة.
- ♦ يستعمل الاستثناء ثلاث كلمات أساسية هي: try, catch, throw.
- ♦ الخطوات التالية تلخص عملية الاستثناء:-
  - ❑ يتم تنفيذ البرنامج بشكل طبيعي خارج كتلة المحاولة.
  - ❑ ينتقل التحكم إلى كتلة المعالجة.
  - ❑ قد تؤدى عبارة ما في كتلة المحاولة ؟؟؟ خطأ في عضو دالي.
  - ❑ يرمى العضو الدالي استثناء.
  - ❑ ينتقل التحكم إلى كتلة الالتقاط التي تلي كتلة المحاولة.



الأسئلة

1/ أكتب دالة قالب تدعى IsEqualTo والتي تقارن بين وسيطين باستعمال العامل == وترجع 1 إذا كانتا متطابقتين و 0 إذا كانتا غير ذلك.
ثم أكتب برنامجاً لاختبار هذه الدالة مع أنواع بيانات C++ الأساسية.
قم بتحميل العامل == بشكل زائد واختبرها مع كائنات.

2/ ما هي العلاقة بين قوالب الدالات وتحميل الدالات بشكل زائد.

3/ وضح العلاقة بين قالب الفئات والوراثة.

4/ عرف الإستثناء.

5/ أكتب الخطوات التي توضح عملية الإستثناء.

6/ أكتب برنامجاً تستخدم فيه آلية الإستثناءات.

11.0

الوحدة الحادية عشرة

دفق دخل/خرج C++



بنهاية هذه الوحدة:

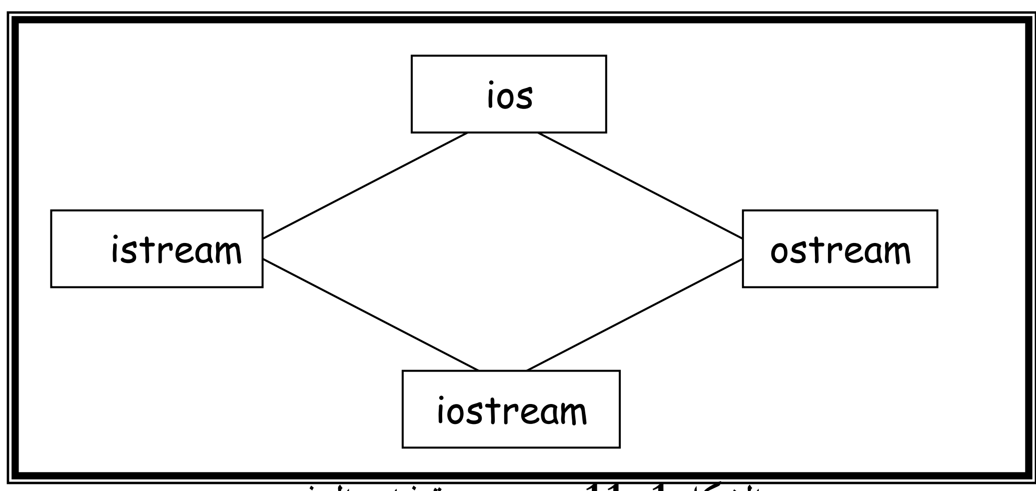
- ستتمكن من استخدام (دفق دخل/ خرج) **(Input/Output Stream)** في لغة **C++**.
- ستتمكن من تنسيق الدخل /الخرج.
- ستتعرف على كيفية إدخال وإخراج الكائنات التي تنشئها بنفسك.
- ستتمكن من إنشاء مناورات خاصة بك.

11.1 مقدمة

الدفق هو اسم عام يطلق لسيل من البيانات في حالة دخل/خرج . يتم تمثيل دفق (الدخل/ الخرج) بكائن تابع لفئة معينة ، فمثلاً رأينا في جميع الأمثلة السابقة كائنات الدفق cin، cout والتي استعملناها لعمليات الدخل والخرج.

11.2 هرمية فئات الدفق

تابع الشكل (1-11) التالي:



الشكل 1-11 يوضح هرمية فئات الدفق

كما نرى من الشكل الفئة **ios** هي الفئة القاعدة لهرمية دفق الدخل والخرج وهى تحتوى على العديد من الثوابت والأعضاء الدالية المشتركة بين مختلف الأنواع من فئات الدخل والخرج. الفئتان **istream** و **ostream** مشتقات من الفئة **ios** وهما متخصصتان بأعمال الدخل والخرج . تحتوى الفئة **istream** على أعضاء دالية لـ **( )get** ، **( )getline** وعامل الدخل(>>) بينما تحتوى الفئة **ostream** على **( )put** و **( )write** وعامل الخرج (<<).

تحتوى الفئة **ios** على أغلبية الميزات التي تحتاج إليها لاستخدام الدفق في ++C ومن أهم هذه الميزات أعلام التنسيق.

11.3 أعلام التنسيق
Format state flags

هي مجموعة من الأعضاء في الفئة ios تعمل لتحديد خيارات في عمل وتنسيق الدخل والخرج.

هنالك عدة طرق لضبط أعلام التنسيق ، وبما أن الأعلام هي أعضاء في الفئة ios يجب عادة وضع اسم الفئة ios وعامل دقة المدى قبلها . يمكن ضبط كل الأعلام باستعمال الأعضاء الدالية ( )setf و( )unsetf التابعة للفئة ios:-

الجدول التالي يبين بعض لأعلام تنسيق الفئة ios:-

العلم	معناه
skipws	تجاهل المسافات البيضاء الموجودة في الدخل
left	محاذاة الخرج إلى اليسار
right	محاذاة الخرج إلى اليمين
dec	تحويل إلى عشري
showbase	استعمال مؤشر القاعدة في الخرج
showpoint	إظهار النقطة العشرية في الخرج
uppercase	استعمال الأحرف الكبيرة في الخرج
showpos	عرض (+) قبل الأعداد الصحيحة الموجبة

البرنامج التالي يوضح كيفية استعمال علمي التنسيق showpos و showpoint:-

```
//Program 11-1:
#include <iostream.h>
int main( )
{
cout.setf(ios:: showpoint);
cout.setf(ios:: showpos);

cout<< 100.0; // displays + 100.0
return 0 ;
```

```
}
```

الخرج من البرنامج:

```
+100.00
```

11.4 المناورات Manipulators

المناورات هي تعليمات تنسيق تدرج في الدفق مباشرة ، رأينا منها حتى الآن المناور endl والثاني يرسل سطراً جديداً إلى الدفق.

هنالك نوعان من المناورات ، نوع يأخذ وسيطة والآخر لا يأخذ أي وسيطة، الجدول التالي يوضح بعض المناورات التي لا تأخذ أي وسيطات:-

المناور	هدفه
ws	تنشيط ميزة تخطى المسافات البيضاء الموجودة في الداخل
dec	التحويل إلى عشري
oct	التحويل إلى ثماني
hex	التحويل إلى ست عشري
endl	إدراج سطر جديد
ends	إدراج حرف خامد لإنهاء سلسلة خرج

تدرج هذه المناورات في الدفق مباشرة ، فمثلاً لخرج المتغير var في التنسيق الستعشرى نكتب:

```
cout<<hex<<var;
```

إن الحالة التي تضبطها المناورات ليس لها وسيطات تبقى نشطة إلى أن يتم تدمير الدفق وعليه يمكننا خرج عدة أرقام في التنسيق الستعشرى من خلال إدراج مناور hex واحد فقط.

الجدول التالي يلخص بعض المناورات التي تأخذ وسيطات ونحتاج إلى إدراج ملف الترويسة iomanip.h لكي نستعمل هذه المناورات:-

المناور	الوسيطة	هدفه
setw()	عرض الحقل (int)	ضبط عرض الحقل المطلوب عرضه
setfill()	حرف الحشو (int)	ضبط حرف الحشو في الخرج(الحرف الافتراضي هو المسافة
setprecision()	الدقة (int)	ضبط الدقة(كمية الأرقام المعروضة)
set iosflags()	أعلام تنسيق (long)	ضبط الأعلام المحددة
Resetiosflags()	أعلام تنسيق (long)	مسح الأعلام المحددة

إن المناورات التي تأخذ وسيطات تؤثر فقط على البند التالي في الدفق فمثلاً إذا استعملنا المناور setw( ) لضبط عرض الحقل الذي يتم إظهار رقم ما فيه سنحتاج إلى استعماله مجدداً مع الرقم التالي.

المثال التالي يستعمل بعض هذه المناورات :

```
//Program 11-2:
#include <iostream.h>
#include <iomanip.h>

int main( )
{
cout << hex << 100 << endl;
cout << setfill('?') << setw(10) << 2343.0;
return 0;
}
```

الخرج من البرنامج:

```
64
??????2343
```

11.5 دوال الدفق
Stream Functions

تحتوى الفئة ios على عدد من الدالات التي يمكن استخدامها لضبط أعلام التنسيق وتنفيذ مهام أخرى . الجدول التالي يبين معظم هذه الدالات .

الدالة	هدفها
ch=fill( );	إعادة حرف الحشو(الفراغ هو الافتراضي)
fill(ch);	ضبط حرف الحشو
p=precision( );	الحصول على الدقة
precision(p);	ضبط الدقة
w=width( );	الحصول على عرض الحقل التالي
setf(flags);	ضبط أعلام التنسيق المحددة
unsetf (flags);	إلغاء ضبط أعلام التنسيق المحددة
setf(flags,field);	مسح الحقل أولاً ثم ضبط الأعلام

يتم استدعاء هذه الدالات بواسطة كائنات الدفق باستعمال عامل النقطة ،فمثلاً لضبط عرض الحقل عند 5 يمكننا كتابة :

```
cout.Width(5);
```

أيضاً تضبط العبارة التالية حرف الحشو عند * :-

```
cout.fill('*');
```

البرنامج التالي يستخدم الدوال width( ) وprecision( ) وfill( ) .

```
//Program 11-3:
#include <iostream.h>
#include <iomanip.h>
int main( )
{
cout.precision (4) ;
cout.width(10);
cout<< 10.12345 <<"\n" ;
cout<<setfill('?');
cout.width(10);
```

```
cout<< 10.12345 <<"\n" ;
//Continued
// field width applies to strings, too
cout.width(10);
cout<< " Hi!" <<"\n" ;
cout.width(10);
cout.setf(ios::left);
cout<< 10.12345  ;
return 0;
}
```

الخرج من البرنامج:

```
        10.12
*****10.12
*******Hi!
10.12*****
```

11.6 الفئة istream

تنفذ الفئة **istream** المشتقة من الفئة **ios** نشاطات خاصة بالدخل ونشاطات إضافية. الجدول التالي يوضح بعض دالات الفئة **istream** .

الدالة	هدفها
>>	إدخال منسق لكل الأنواع الأساسية والمحملة بشكل زائد
get(ch)	إدخال حرف واحد
get(str)	إدخال أحرف إلى مصفوفة وصولاً إلى '\o'
get(str,max)	إدخال حتى max أحرف إلى المصفوفة
peek(ch)	قراءة حرف واحد وتركه في الدفق

putpack(ch)	إعادة إدراج الحرف الأخير المقروء في دفق الدخل
count=gcount	إعادة عدد الأحرف التي قرأها استدعاء الدالة get() و getline()

لقد رأينا حتى الآن بعضاً من هذه الدالات كـ get( ) مثلاً .معظمها يعمل على الكائن cin الحقيقي يمثل لوحة المفاتيح.

11.7 الفئة ostream

تعالج الفئة **ostream** نشاطات الخرج، يبين الجدول التالي أغلب الدالات التي تستعملها هذه الفئة:-

الدالة	هدفها
<<	إخراج منسق لكل الأنواع الأساسية والمحملة بشكل زائد
put(ch)	إخراج الحرف **ch** في الدفق
flush( )	مسح محتويات الدارئ **(Buffer)** وإدراج سطر جديد
write(str,size)	إخراج **size** أحرف من المصفوفة **str**

لقد استعملنا حتى الآن كائني دفق **cin**و **cout** . يرتبط هذان الكائنان عادة بلوحة المفاتيح والشاشة على التوالي . هناك كائنان آخران هما **cerr** و **clog** .

غالباً ما يتم استعمال الكائن **cerr** لرسائل الخطأ. الخرج المرسل إلى **cerr** يتم عرضه فوراً ولا يمكن تغيير وجهته لذا ترى رسالة الخرج من **cerr** في حال تعطل البرنامج كلياً. هنالك كائناً مماثلاً لـ **cerr** هو **clog** لكن يتم وضع خرج الكائن في الدارئ على عكس **cerr**

11.8 تحميل العوامل << و >> بشكل زائد

يمكن تحميل العوامل << و >> بشكل زائد لإدخال وإخراج كائنات تابعة لفئات عرفها المستخدم .البرنامج التالي يقوم بتحميل عامل الإخراج <<بشكل زائد وذلك لإخراج كائن تابع للفئة **phonebook**.

```
//Program 11-4:
#include <iostream>
#include <cstring>

class phonebook {
// now private
```

```
char name[80];
int areacode;
//Continued
int prefix;
int num;
public:
phonebook(char *n, int a, int p, int nm)
{
strcpy(name, n) ;
areacode = a;
prefix =p;
num = nm;
}
friend ostream & operator <<(ostream &stream, phonebook
o);
};
// Display name and phone number.
ostream & operator << (ostream &stream, phonebook o)
{
stream<< o.name <<" ";
stream << "(" << o.areacode << ") " ;
stream <<o.prefix<< "-" << o.num <<"\n" ;
return stream;  // must return stream
}
int main( )
{
phonebook a("Mohammed", 011, 011, 123456);
phonebook b("Omer" , 031, 011, 576890);
phonebook c("Ali" , 261, 011, 999009);
cout<<a<<b<<c;
return 0;
}
```

الخرج من البرنامج:

```
Mohammed  (011) 011 -123456
Omer (031) 011-576890
Ali (261) 011- 999009
```

لاحظ في الدالة **main( )** مدى سهولة معاملة كائنات الفئة **phonebook** كأي نوع بيانات أساسي آخر باستعمال العبارة:-

```
cout<<a<<b<<c;
```

تم تعريف الدالة **operator<< ( )** على أنها صديقة للفئة **phonebook** وذلك لأن كائنات **ostream** تظهر في الجهة اليسرى للعامل وهى تفيد كائناً تابعاً للفئة **ostream** (العامل <<)، تسمح قيم الإعادة هذه خرج أكثر من قيمة واحدة في العبارة . ينسخ العامل << البيانات من الكائن المحدد كالوسيطة الثانية ويرسلها إلى الدفق المحدد كالوسيطة الأولى.

تحميل العامل >> بشكل زائد:-

وبنفس الطريقة يمكننا تحميل العامل >> بشكل زائد لإدخال الكائنات التي يعرفها المستخدم بنفسه. البرنامج التالي يسمح للمستخدم باستعمال العامل >> لإدخال كائنات تابعة للفئة **phonebook** .

```
//Program 11-5:
#include <iostream.h>
#include <cstring.h>

class phonebook  {
char name[80];
int areacode;
int prefix;
int num;
```

```
public:
phonebook( ) { };
phonebook(char *n, int a, int p, int nm)
{
strcpy(name, n) ;
areacode = a;
//Continued
prefix =p;
num = nm;
}
friend ostream & operator<<(ostream &stream, phonebook
o);
friend istream & operator>>(istream &stream, phonebook
&o);

};
// Display name and phone number.
ostream & operator << (ostream &stream, phonebook o)
{
stream<< o.name <<" ";
stream << "(" << o.areacode << ") " ;
stream <<o.prefix<< "-" << o.num <<"\n" ;
return stream;  // must return stream
}
// Input name and telephone number.
istream & operator>> (istream &stream, phonebook &o)
{
cout << " Enter name: ";
stream>> o.name;
cout << " Enter area code: ";
stream>> o.areacode;
cout << " Enter prefix: ";
stream>> o.prefix;
cout << " Enter number: ";
```

```
stream>> o.num;
cout<<"\n" ;
return stream;
}
int main( )
{
phonebook b;
cin>> b;
cout << b;
//Continued
return 0;
}
```

الخرج من البرنامج:



```
Enter name:  Ahmed
Enter area code:  111
Enter prefix:  555
Enter number:  1010

Ahmed(111)555 -1010
```

11.9 كيفية إنشاء مناورات خاصة بنا

يمكن أيضاً للمستخدم إنشاء مناورات تقوم بتنسيق خاص بالمستخدم .

الصورة العامة لإنشاء مناور خرج هي:-

```
ostream & mani-name( ostream & stream)
{
//your code here
return stream;
}
```

المثال التالي يقوم بإنشاء مناورين ( )la و( )ra يقومان بإخراج (→) و (←).

```
//Program 11-6:
#include <iostream>
#include <iomanip>
#include <conio.h>

// Right Arrow
ostream &ra(ostream &stream)
{
stream << "-> " ;
return stream;
}
// Left Arrow
ostream &la(ostream &stream)
{
stream << "<- " ;
return stream;
}
int main( )
{
cout << "High balance" <<ra<< 1233.23<<"\n";
cout <<"Over draft" << ra<<567.66<< la;
getch();
return 0;
}
```

الخرج من البرنامج:

```
High balance  → 1233.23
Over draft    → 567.66 ←
```

الصورة العامة لإنشاء مناور دخل هي:-

```
istream & mani-name(istream & stream)
{
//your code here
return stream;
```

}

المثال التالي يقوم بإنشاء مناور دخل getpass( ) والثاني يقوم بإخراج صوت جرس باستعمال تتابع الهروب '\a' ويطلب من المستخدم إدخال password .

```
//Program 11-7:
#include <iostream>
#include <cstring>

// Asimple input manipulator.
istream &getpass (istream &stream)
{
cout << '\a' ; // sound bell
cout << "Enter password: ";
return stream;
}
int main( )
{
char pw[80];

do cin>> getpass >>pw;
  while (strcmp (pw, "password"));
cout <<"logon complete\n";
return 0;
}
```

الخرج من البرنامج:



```
Enter password: password
Login complete
```



الملخص:

- الدفق هو اسم عام يطلق لسيل من البيانات في حالة دخل /خرج.
- الفئة **ios** هي الفئة القاعدة لهرمية دفق الدخل / الخرج.
- الفئات **istream**، **ostream** مشتقتان من الفئة **ios** وهما مختصتان بأعمال الدخل والخرج.
- أعلام التنسيق هي مجموعة من الأعضاء في الفئة **ios** تعمل على تنسيق الدخل والخرج.
- المناورات هي تعليمات تنسيق تدرج في الدفق مباشرة.
- هنالك نوعان من المناورات، نوع يأخذ وسيطة والآخر لا يأخذ أي وسيطة.
- الحالة التي تضبطها المناورات التي ليس لها وسيطات تبقى نشطة إلى أن يتم الدفق.
- عند استعمال المناورات يجب إخراج ملف الترويسة **iomanip.h**.
- تحتوى الفئة **ios** على عدد من الدالات التي يمكن استخدامها لضبط أعلام التنسيق.
- تنفذ الفئة **ostream** المشتقة من الفئة **ios** نشاطات خاصة بالدخل.
- تعالج الفئة **ostream** نشاطات الخرج.
- يتم استعمال الكائن **cerr** لعرض رسائل الخطأ.
- يمكن تحميل **>>** و **<<** بشكل زائد لإدخال و بشكل زائد لإدخال وإخراج كائنات تابعة لفئات عرفها المستخدم.
- يمكن إنشاء مناورات تقوم بتنسيق خاص بالمستخدم.



الأسئلة

١ – قم بكتابة برنامج ينفذ الآتي:

- طباعة العدد الصحيح 40000 مع محاذاته على اليسار على أن يكون عرض الحقل 15.
- قراءة سلسلة وتخزينها في مصفوفة أحرف state.
- طباعة 200 بعلامة وبدون علامة.
- طباعة العدد 100 بالنظام السادس عشر.

2/ أكتب برنامجاً لدخل أعداد صحيحة بالنظام العشري والثماني والسادس عشر وخرج هذه الأعداد. اختبر البرنامج

بالبيانات الآتية:

0×10 ,   010   ,   10

12.0

الوحدة الثانية عشرة

معالجة الملفات File Processing



بنهاية هذه الوحدة:

- ستتمكن من التعامل مع الدفق وتتعرف على الملفات التتابعية.
- ستتمكن من إنشاء ملفات تتابعية، والتعامل معها.
- ستتمكن من الوصول إلى السجلات المختلفة تتابعياً.
- ستتمكن من الوصول إلى السجلات المختلفة عشوائياً.

12.1 مقدمة

تخزين البيانات في المتغيرات أو المصفوفات هو تخزين مؤقت، لذلك نحتاج الى وسيلة تخزين دائمة. وتوفر الملفات Files هذه الوسيلة.

يخزن الحاسوب الملفات في وسائط التخزين الثانوية مثل الأقراص.

في هذه الوحدة، سنوضح كيفية إنشاء ومعالجة الملفات من خلال برامج لغة C++.

عادة تتكون المفات من مجموعة من السجلات Records والتي تتكون بدورها من مجموعة من الحقول Fields. يتكون ملف للموظفين مثلاً على مجموعة من السجلات (سجل لكل موظف)، وقد يحتوي السجل مثلاً على الحقول التالية:

١. رقم الموظف.

٢. إسم الموظف.

٣. العنوان.

٤. المرتب.

لتسهيل الوصول الى سجل ما في ملف، يتم اختيار حقل مفتاحي للسجل Record Key . والذي يجب أن يكون فريداً Unique في الملف.

في ملف الموظفين اعلاه، يمكن اختيار رقم الموظف كحقل مفتاحي للملف.

هناك عدة طرق لتنظيم السجلات داخل الملف، أشهر الطرق المستخدمة هي الملفات التتابعية Sequential Files والتي يتم فيها تخزين السجلات بترتيب حقولها المفتاحية، فمثلاً في ملف الموظفين، يكون أول سجل هو السجل الذي يحمل أقل رقم موظف.

12.2 الملفات والدفق Files and Stream

تعامل C++ الملفات كفيض متتابع من الثمانيات Bytes. الشكل التالي يوضح ملف يتكون من n Byte

0	1	2	3	4	5	6	7	8	9		n-1	
										....		end

عند فتح ملف يتم إنشاء كائن يقترن معه الدفق. لقد رأينا من قبل أربعة كائنات منشأة أتوماتيكياً، وهي cout، cin ، cerr و clog.

يستخدم الكائن cin لإدخال بيانات من لوحة المفاتيح، والكائن cout يستخدم لإخراج بيانات إلى الشاشة، والكائنان cerr وclog يستخدمان لإخراج رسائل الأخطاء إلى الشاشة.

عند التعامل مع الملفات، يجب تضمين ملفي الترويسة iostream.h و fstream.h حيث يحتوي الملف fstream.h على فئات الدفق ifstream (والتي تستخدم في إدخال بيانات إلى الملفات) و ofstream (والتي تستخدم لإخراج بيات من المفات)، و fstream (لإدخال وإخراج بيانات من الملفات).

لفتح ملف، نحتاج لإنشاء كائن يتبع لإحدى هذه الفئات.

12.3 إنشاء ملف تتابعي Creating a Sequential file

لا تتطلب C++ أي هيكلية معينة للملف، وعليه لا يوجد مصطلح سجلات في ملفات C++ لذا يجب على المبرمج تحديد الكيفية التي يتم بها تنظيم الملف.

البرنامج التالي يوضح كيفية إنشاء ملف تتابعي:

```
//Program 12-1
//Creating a sequential file
#include<iostream.h>
#include<fstream.h>
#include<stdlib.h>
main( )
{
 ofstream   outclientfile("clients.dat",ios::out);
 if (!outclientfile){
    cerr<<"File could not be opened"<<endl;
    exit (1);
}
  cout<<"Enter the account, name, and balance."
        <<endl
```

```
        <<"(Enter EOF to end input)"<<endl
        <<"? ";
  int account;
  char name[10];
//Continued
  float balance;
while(cin>>account>>name>>balance){
outclientfile<<account<<" "<<name<<" "<<balance
             <<endl;
cout<<"? ";
}
 return 0;
}
```

الخرج من البرنامج:

```
Enter the account, name, and balance.
(Enter EOF to end input)
? 100 Ahmed 24.98
? 200 Ali 345.67
? 300 Hassan 0.00
? 400 Omer -42.16
? 500 Abbas 224.62
? ^Z
```

البرنامج السابق ينشئ ملفاً تتابعياً، حيث يمكن استخدامه في نظام حسابات مثلاً ليساعد في إدارة حسابات العملاء.

لكل عميل من العملاء، يتحصل البرنامج على رقم حساب العميل **account** وإسم العميل **name** ورصيد العميل **balance**. البيانات التي يتحصل عليها البرنامج لكل عميل تمثل سجل ذلك العميل.

يستخدم رقم حساب العميل كحقل مفتاحي، وعليه يكون الملف مرتباً بترتيب أرقام حسابات العملاء.

تم فتح الملف للكتابة فيه، لذلك ينشئ البرنامج كائن خرج تابع للفئة **ofstream** يدعى **outclientfile**، وتم تمرير وسيطتين لمشيد ذلك الكائن وهما إسم الملف **Clients.dat**، طريقة فتح الملف (File open mode) ios::out

يقوم البرنامج، باستقبال البيانات المدخلة وحفظها في الملف، إلى أن يتم إدخال رمز نهاية الملف (<**ctrl**> Z).

خرج البرنامج يفترض أنه تم إدخال بيانات خمسة عملاء، ثم تم إدخال رمز نهاية الملف **^Z**.

نلاحظ أننا قمنا بتضمين ملف الترويسة **stdlib.h** الذي يحتوي على تعريف الدالة **exit**، والتي تنهي البرنامج في حالة عدم فتح الملف بصورة صحيحة.

12.4 قراءة البيانات من ملف تتابعي Reading Data from a Sequential file

سنقوم الآن بكتابة برنامج يقوم بقراءة الملف السابق، وطباعة محتوياته على الشاشة:

```
Program 12-2:
//Reading and printing a Sequential file
#include<iostream.h>
#include<fstream.h>
#include<iomanip.h>
#include<stdlib.h>

void outputline(int, char *, float);
main( )
{
  ifstream inClientFile("clients.dat",ios::in);
```

```
  if (!inClientFile) {
      cerr << "File could not be opened" <<endl;
      exit(1);
   }
int account;
char name[10];

//Continued
float balance;
cout   <<setiosflags(ios::left) <<setw(10) <<"Account"
       <<setw(13) <<"Name" <<"Balance"<<endl;

while(inClientFile >> account >.name >>balance)
 outputline(account, name, balance);
 return 0;
}
void outputline(int acct, char *name, float bal)
{
       cout << setiosflags(ios::left) << setw(10)<< acct
             << setw(13) << name<< setw(7)
             << setprecision(2)
              << setiosflags(ios::showpoint | ios::right)
              << bal << endl;
}
```

الخرج من البرنامج:

Account	Name	Balance
100	Ahmed	24.98
200	Ali	345.67
300	Hassan	0.00
400	Omer	-42.16
500	Abbas	224.62

يتم فتح الملفات لقراءة بيانات منها بإنشاء كائن يتبع للفئة **ifstream** والذي يتم تمرير وسيطتين له هما إسم الملف **clients.dat** وطريقة فتح الملف **File Open mode**. فالإعلان:

```
ifstream inClientFile("clients.dat",ios::in);
```

ينشئ كائن تابع للفئة **ifstream** يدعى **inClientFile**، ليقوم بفتح الملف **clients.dat** للقراءة منه.

12.5 الوصول العشوائي لمحتويات ملف تتابعي Random Access to a Sequential file

يملك كل كائن ملف، مؤشرين مقترنين به يسميان مؤشر الحصول **get pointer** ومؤشر الوضع **put pointer**، ويسميان أيضاً مؤشر الحصول الحالي ومؤشر الوضع الحالي.

في بعض الأحيان، قد نرغب في بدء قراءة الملف من بدايته ومتابعته إلى نهايته، وقد ترغب عند الكتابة البدء من البداية وحذف أي محتويات موجودة ،لكن هنالك أوقات نحتاج فيها إلى التحكم بمؤشرات الملفات. لكي نتمكن من القراءة أو الكتابة في مواقع عشوائية من الملف.

تتيح الدالتان **seekg** و **seekp** ضبط مؤشري الحصول والوضع على التوالي.

يمكن استخدام الدوال **( )seekg** و **( )seekp** بطريقتين :-

1/ مع وسيطة واحدة هي موقع البايت المطلق في الملف (بداية الملف هي البايت **0**).

2/ مع وسيطتين الأولى إزاحة من موقع معين في الملف والثانية الموقع الذي تم قياس الإزاحة منه. هنالك ثلاثة احتمالات للوسيطة الثانية:-

(أ) **beg** وهى بداية الملف.

(ب) **Cur** وتعنى الموقع الحالي للمؤشر.

(ت) **End** وتعنى نهاية الملف.

فمثلاً العبارة :-

```
seekp( -10,ios:: end);
```

ستضع مؤشر الوضع **10** بايتات قبل نهاية الملف.

البرنامج التالي يستخدم الدالة **seekg** مع وسيطة واحدة:

Program 12-3:

```
#include<iostream.h>
#include<fstream.h>
#include<iomanip.h>
#include<stdlib.h>
void outputline(int, char *, float);
main( )
{
//Continued
  ifstream inClientFile("clients.dat", ios::in);
  if (!inClientFile){
     cerr<< "File could not be opened" <<endl;
     exit(1);
  }
cout <<"Enter request "<<endl
        <<" 1 – List accounts with zero balances" << endl
        <<" 2 – List accounts with credit balances"<<endl
        <<" 3 – List accounts with debit balances"<<endl
        <<" 4 – End of run" <<endl<<"?";
int request;
//Continued
cin>> request;
while(request !=4) {
        int account;
        char name[10];
        float balance;
        inClientFile >>account >>nam>>balance;
        switch (request) {
           case 1:
              cout <<endl <<"Account with zero balances:"
                       <<endl;
             while(!inClientFile.eof()) {
              if (balance==0)
                 outputline(account, name, balance);
```

```
            inClientFile >>account >>name >>balance;
        }
        break;
        case 2:
          cout<<endl<<"Accounts with credit balance:"
                <<endl;
         while(!inClientFile.eof()) {

          if (balance <0)
             outputline(account, name, balance);
        //Continued
            inClientFile>>account >>name >>balance;
        }
        break;
        case 3:
          cout<<endl<<"Accounts with debit balances:"
               <<endl;
           while(!inClientFile.eof()) {
             if (balance > 0)
              outputline(account, name, balance);
        inClientFile >>account>>name>>balance;
        }
        break;
}
  inClientFile.clear( );   //reset eof for next input
  inClientfile.seekg(0); //position to beginning of file
  cout<<endl <<"? ";
  cin>>request;
}

cout << "End of run." <<endl;

  return 0;
}
```

```
cout << setiosflags(ioa::left) << setw(10) << acct
       << setw(13) << name <<setw(7) <<setprecision(2)
       << setiosflags(ios::showpoint | ios::right)
        <<bal <<endl;
}
```

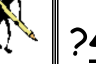
الخرج من البرنامج:

Enter request
1 – List accounts with zero balances
2 – List accounts with credit balances
3 – List accounts with debit balances
4 – End of run
?**1**
Accounts with zero balances:
300 Hassan 0.00

?**2**

Accounts with credit balances:
400 Omer -42.16

?**3**
Accounts with debit balances:
100 Ahmed 24.98
200 Ali 345.67
500 Abbas 224.62



?**4**

End of run.



الملخص:

- الملفات هي وسيلة دائمة لتخزين البيانات.
- تتكون الملفات عادة من مجموعة من السجلات.
- تتكون السجلات من مجموعة من الحقول.
- يكون لكل سجل حقل مفتاحي.
- في الملفات التتابعية يتم تخزين السجلات بترتيب حقولها المفتاحية.
- عند التعامل مع الملفات يجب تضمين الملف **fstream.h**.
- عند فتح ملف للكتابة فيه يجب إنشاء كائن تابع للفئة **ofstream**.
- يتم فتح الملفات لقراءة بيانات منها بإنشاء كائن يتبع الفئة **ifstream**.
- لإسترجاع بيانات من ملف تتم قراءة الملف من بدايته وقراءة كل محتويات الملف بالتتابع حتى نصل إلى البيانات المطلوبة.
- يملك كل كائن ملف مؤشرين مقترنين به يسميان مؤشر الحصول **get pointer** ومؤشر الوضع **Put pointer**.
- تضبط الدالتان **seekg( )** و **seekp( )** مؤشري الحصول والوضع على التوالي.



الأسئلة

١ – أنشئ ملف للموظفين يدعى **Employee** على أن يحتوي كل سجل في الملف على الحقول التالية:–

- رقم الموظف.
- إسم الموظف.
- العنوان.

ثم قم بإدخال بيانات خمسة موظفين.

٢ – تأكد من إدخال البيانات في السؤال السابق بصورة صحيحة وذلك بكتابة برنامج لقراءة محتويات الملف.

٣ – قم بكتابة برنامج يقوم باستقبال معلومات عن طلاب كلية ويضعها في ملف يسمى **Students**، بحيث يحتوي ملف الطلاب على الآتي:

- رقم الطالب.
- إسم الطالب.
- تخصص الطالب.
- درجة الطالب.
- ومن ثم قم بكتابة برنامج يقوم بقراءة هذا الملف.



مع تحيات: محمد الغشيمي
جامعة الحديدة
كلية علوم وهندسة الحاسوب
مستوى ثاني
E-mail. Mohmmed_apdo2012@hotmail.com